NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

# ANALYSIS FOR THE PURPOSE OF DEVELOPING COURSE MATERIAL FOR INSTRUCTING GRADUATE STUDENTS IN OBJECT ORIENTED PROGRAMMING WITH ADA 95

by
John T. Drimousis
March 1995

Thesis Advisor: David Gaitros

| REPORT DOCUMENTATION PAGE | | | Form Approved OMB No. 0704 |
|---|---|---|---|

| | | |
|---|---|---|
| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE March 1995 | 3. REPORT TYPE AND DATES COVERED Master's Thesis |

| | |
|---|---|
| 4. TITLE AND SUBTITLE ANALYSIS FOR THE PURPOSE OF DEVELOPING COURSE MATERIAL FOR INSTRUCTING GRADUATE STUDENTS IN OBJECT ORIENTED PROGRAMMING WITH ADA 95 | 5. FUNDING NUMBERS |
| 6. AUTHOR John T. Drimousis | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000 | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER |

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| | |
|---|---|
| 12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited | 12b. DISTRIBUTION CODE |

13. ABSTRACT *(maximum 200 words)*

The Department of Computer Science located at the Naval Postgraduate School, Monterey, California recently decided to teach the new version of ADA 95 in their beginning programming class (CS 2972). The problem was two fold: (1) Teaching a newly altered language incorporating features from the old version into the new while retaining forward compatibility and (2) presenting object oriented design and programming features to students who have little or no programming experience.

What evolved as the best method was to postpone the introduction of object-oriented design until the latter half of the class. Highlights and differences of the languages would be presented during the course of the class with emphasis placed an old features that should be avoided that would make forward compatibility difficult. What resulted was a complete restructuring of the course.

| | | | |
|---|---|---|---|
| 14. SUBJECT TERMS object-oriented programming, class wide programming, dispatching, late binding | | | 15. NUMBER OF PAGES 424 |
| | | | 16. PRICE CODE |
| 17. SECURITY CLASSIFICATION OF REPORT Unclassified | 18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified | 19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified | 20. LIMITATION OF ABSTRACT UL |

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

# ANALYSIS FOR THE PURPOSE OF DEVELOPING COURSE MATERIAL FOR INSTRUCTING GRADUATE STUDENTS IN OBJECT ORIENTED PROGRAMMING WITH ADA 95

by
John T. Drimousis
Lieutenant, Hellenic Navy

Submitted in partial fulfillment
of the requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**March 1995**

Author: [illegible]
John Drimousis

Approved by: [illegible]
David A. Gaitros, Advisor

[illegible]
Lucia Luqi, Co-Advisor

[illegible]
Ted Lewis, Chairman
Department of Computer Science

## ABSTRACT

The Department of Computer Science located at the Naval Postgraduate School, Monterey, California recently decided to teach the new version of ADA 95 in their beginning programming class (CS 2972). The problem was two fold: (1) Teaching a newly altered language incorporating features from the old version into the new while retaining forward compatibility and (2) presenting object oriented design and programming features to students who have little or no programming experience.

What evolved as the best method was to postpone the introduction of object oriented design until latter half of the class. Highlights and differences of the languages would be presented during the course of the class with emphasis placed on old features that should be avoided that would make forward compatibility difficult. What resulted was a complete restructuring of the course.

## TABLE OF CONTENTS

## LIST OF FIGURES

# I. INTRODUCTION

## A. USE OF ADA IN THE COMPUTER SCIENCE DEPARTMENT

The Department of Computer Science at the Naval Postgraduate School (NPS) basis there Software Engineering and several other curriculum tracks are based on the Department of Defenses' ADA programming language. With the development of a new version of ADA (ADA 95) and the addition of the Object Oriented paradigm, a new method of teaching ADA to beginning computer scientists was required.

In addition to teaching the new language to beginning students the Department of Computer Science is required to continuously improve their curriculum to offer the best possible course of study to the students and to keep up with their technology. Object Oriented programming is gaining popularity as a very useful means of developing more reliable software. ADA 95 has all of the properties required by todays software developers.

The policy established by the DoD requires all software possible should be developed using the new ADA95 and the Department of Computer Science at NPS is committed to supporting this policy. This thesis will lay the framework for all future ADA95 course development and implementation.

## B. HISTORY OF ADA

In 1974, the Department of Defense (DoD) published a report estimating the future costs of its software at the horrendous amount of over $3 billion annually. In addition, there are hundreds of languages or dialects being used by the DoD and its contractors, making it difficult to interchange programs, programmers and virtually impossible for effective software maintenance. The services to minimize the software cost and eliminate all these serious problems, the same year, proposed to DoD the adoption of a common language. ( Sammet, 1986, pp. 723-729)

Early in 1977 the set of requirements called IRONMAN has been completed and the DoD held a design competition. Seventeen proposals were received, and four were chosen to go ahead in parallel and in competition. The four contractors with their colour coding were, Cii-Honeywell Bull ( Green ), Intermetrics ( Red ), SofTech( Blue ) and SRI International ( Yellow ) . In early 1978 the initial designs were ready and the DoD judged that the Cii-Honeywell Bull and Intermetrics designs showed more promise than the other two which are eliminated. The final designs, based on the final set of requirements, called STEELMAN were sent out for public commentary. The final choice of the language was made on 2 May 1979 and the design competition's eventual winner was the Cii-Honeywell Bull ( Green ) team, led by the Frenchman Jean Ichbiah. ( daCosta, 1984, pp. 1-4)

The name Ada was chosen to honor the mathematician Lady Augusta Ada Byron, Countess of Lovelace and daughter of the poet Lord Byron. In early 1983 the Ada language was further modified to reflect in the final changes that led to the ANSI ( American National Standards Institute) standard document.

This new language included facilities offered by classical languages such as Pascal as well as facilities often found only in specialized languages like Simula 67. Thus the Ada has the ability to define types, subprograms to serve the need for modularity, whereby data, types and subprograms can be packaged. In addition to those aspects, the language covers real-time programming, with facilities to model parallel tasks and to handle exceptions facilities that only this language offered only. ( Gonzalez, 1991, pp. 1-1)

## C. ADA 95 GENERATION

Although Ada 83 has all the aspects to provide a single flexible yet portable language for real-time embedded systems to meet the needs of the US DoD, its domain of application had expanded to include many other areas, such as large-scale information systems, distributed systems, large scientific computation, and systems programming. Furthermore, its user base had expanded to include all major defense agencies of the western world, the whole of aerospace community and increasingly many areas in civil and private sectors such as telecommunications, process control and monitoring systems. Indeed, the expansion in the civil sector is such that civil applications now generate the dominant revenues of many vendors. ( Ada 9x Mapping / Revision Team , 1994, pp. 1-1, 1-2)

In 1988, the DoD, to ensure the continuation of Ada's implementation to all the present and future application areas and in addition to ensure that the needs of the whole-world-wide Ada community ( and not just the defense community) were taken into account, initiated a revision of ANSI/MIL-STD 1815A ( Ada 83 ). By December 1990, all the revision requests had been analyzed along with the 850 Ada revised issues which had been submitted between 1983 and 1988 to ISO working group 9. This analysis resulted in the publication of the following new Ada Requirements Document in December 1990 :

**Revise ANSI/MIL-STD-1815A-1983 to reflect current essential requirements with minimum negative impact and maximum positive impact to the Ada community.**

By January 1991, the process of mapping the requirements into proposed language changes was well underway. A mapping / revision team proposes language changes meeting the requirements document. Four prototyping teams ( representing different compilers, computer architecture and applications domain) try out the language changes in their compilers and run specific application software on the modified compiler to fully evaluate impact.

The first phase of the ANSI/ISO approval process was successfully completed in January 31, 1993. In March 1994, ISO delegates discussed technical issues with the Ada 94 team, and the last phase (or standardization process) for Ada 94 as an ANSI, and ISO standard will be accomplished by December 1994 or during the first months in 1995. ( Anderson, 1994, pp. 16)

## D. MAIN FEATURES OF ADA 95

The output of the Mapping / revision team showed that the new Ada or Ada 9X as was named the language, needs more attention - improvement to accomplish the general features that the users need from the revision Ada, in the following four areas: ( Ada 9x Mapping / Revision Team , 1994, pp. 1-3)

### 1. Interfacing

Ada 83 recognizes the importance of being able to interface to external systems by the provision of features such as representation clauses and pragmas. Nevertheless, it is sometimes not easy to interface certain Ada 83 programs to other language systems. A general need was felt for a more flexible approach allowing, for instance, the secure manipulation of references and the passing of procedures as parameters. An example arises when interfacing into GUI's where it is often necessary to pass a procedure as a parameter for call-back.

### 2. Programming By Extension( Object-Oriented Programming)

Although Ada's package and generic capability are an excellent foundation, nevertheless experience with the Object Oriented paradigm in other languages had shown the advantages of being able to extend a program without any modification to existing proven components.

### 3. Program Libraries

The Ada program library brings important benefits by extending the strong typing across the boundaries between separately compiled units. However, the flat nature of the Ada 83 library gave problems of visibility control; for example it prevented two library packages from sharing a full view of a private type. A common consequence of the flat structure was the package become large and monolithic. This hindered understanding and increased the cost of recompilation.

### 4. Tasking

The Ada rendezvous paradigm is a useful model for the abstract description of many tasking problems. But experience had shown that a more static monitor like approach was also desirable for common shared-data access applications. Furthermore the Ada priority model needed revision in order to enable users to take advantage of the greater understanding of scheduling theory which had emerged since Ada 83 was defined.

## E. THE APPROACH

The revision team had to enhance the Ada functionality in Interfacing, Object Oriented Programming, Libraries, and Tasking facilities by following the rule, to make the minimum corrections to the existing Ada and so to maintain the compatibility. The revision team  had an already big advantage the experience in the last ten years on Ada. After a careful analysis of the limitations that were offered by the Ada 83 a new compatible Ada was produced which implements all the new needs that are desired both by the DoD and the civilian world. ( Anderson, 1992, pp. 35) .

The new Ada has the ability to interface with other systems and programming languages like C, COBOL, and Fortran. The approach to accomplish those new features in the interfacing was by the introduction of the new forms of access types, pragmas, and the following new predefined language packages; (1) **Interfaces** as the parent package and its children (1a) **Interfaces.C** ( handles all the interfacing with C language) (1b)**Interfaces.COBOL** ( handles all the interfacing with COBOL language) and (1c) **Interfaces.Fortran** ( handles all the interfacing with Fortran language) .

The approach to accomplish the object oriented programming features was by the introduction of a programming by extension ( tagged types), a Class wide programming ( Classes), Abstract types and Subprograms, and the Access types (access to subprograms).

The approach to improve the library facilities and the Programming in Large systems was by the introduction of the Hierarchical Libraries which are implemented by the new language features like child packages, private children and generic children.

The Ada 95 overcomes the problems in Tasking by the introduction of the protected types. A protected type encapsulates and provides synchronized access to the private data of objects of the type without the introduction of additional tasks.

## F. OBJECTIVES

The Computer Science Department at NPS, based on  the DoD's policy, has decreed Ada as the basic programming language moreover, Ada 95 is a modern programming language that includes all the capabilities that users desire like object oriented programming, real time capabilities, good Interfacing with other languages, so there is an urgent need to develop a course material for instructing computer science students in the new Ada and its new features in object oriented programming.

The primary emphasis of this thesis is the introduction of the Ada 95 and the construction of a course outline which must be built to endowed the computer science's students with the appropriate knowledge to face the future necessities in all areas of computer science .

The research of this thesis involved with an analysis in an overall Ada 95 core language and particularly of object oriented programming, hierarchical libraries, exceptions, and generics features. In addition we summarize the incompatibilities between Ada 83 and Ada 95 and we present some guidelines to Ada 83 programmers to avoid some common errors when they use Ada 95. We proposed a course outline, material, suitable to introduce the new Ada based on the computer science department's demands.

## G. BACKGROUND

The first priority rule that the Naval Postgraduate School has been constituted to accept a student to study at Computer Science Department is that the students have to know or have experience on at least one programming language like C, Pascal, Fortran, etc. The new course outline for a Ada 95 is constructed based on the above policy. The students background in another programming language lead us to construct the course outline material by assuming that the registered students for this new course have the basic knowledge or can at least understand the common vocabulary of all programming languages like compiler, compilation, linking, types, assignment, iteration, etc. In addition, we try to refresh the students basic knowledge / vocabulary in each step we present the core Ada 95 language.

We used the GNAT compiler Version 1.82 for UNIX to solve the programming examples and case studies which we chose to present both in Ada 95 Highlights and the course outline material.

## H. ORGANIZATION OF THESIS

This chapter presents an introduction to new Ada 95. Chapter II of this thesis provides the main features of Ada 95 which are the object oriented programming, hierarchical libraries, exceptions, and generics . Chapter III discuss the incompatibilities between Ada 83 and Ada 95 and gives guidelines to Ada 83 programmers how to avoid errors when they use Ada 95 . Chapter IV presents the course outline. Chapter V presents the accomplishments and proposals. Appendix A  presents the slides for the core Ada 95 language material. Appendix B includes the slides for the Object Oriented Programming features of Ada 95.

# II. HIGHLIGHTS OF ADA 95

The great strength of Ada 83 is its reliability. The strong typing and related features ensure that programs contain few surprises; most of errors are detected at compile time and of those remaining many are detected by runtime constraints. Moreover the compile-time checking extends across compilation unit boundaries.

However, after a number of years experience it became clear that some improvements were necessary in order to completely satisfy the present and the future user's needs from a whole variety of application areas. As a consequence, Ada 95 was constructed to satisfy all the modern necessities of programming, like objects oriented programming capabilities, real time features, good interfacing etc. The approach was followed by the mapping team to implement the object oriented programming features, was a prototype idea and different from the usual approaches that all the present object oriented programming languages use. The new Ada was constructed without incurring the pervasive overheads of languages such as Smalltalk or the insecurity brought by the weak C foundation in the case of C++. Ada 95 remains a very strongly typed language but provides the prime benefits of all key aspects of the Object oriented paradigm.( Barnes, 1994, pp. 398-400)

Another area with major changes in Ada 95 is the tasking model where the introduction of protected types allows a more efficient implementation of standard problems of shared data access. This brings the benefits of speed provided by low-level primitives such as semaphores without the risks incurred by the use of such unstructured primitives. Moreover, the clearly data oriented view brought by protected types fits in naturally with the general spirit of the object oriented paradigm.

Ada 95 also introduces the hierarchical library structure containing child packages and child subprograms which solve the flat library structure that was used by Ada 83. This approach gives the user the ability to expand a package without recompiling it.

Another area which sees considerable change in Ada 95 is the generic packages. The generics expanded to match the object oriented facilities and the additional numeric types which are offered by the Ada 95. In addition new Ada continues to offer the traditional strength by implementing generics packages.

Ada 95 continues to provide the exception facilities, and introduces a new idea of an exception occurrence that identifies both the exception and the instance of its being raised.

## A. OBJECT ORIENTED PROGRAMMING

Objected-oriented programming has become exceedingly popular in the past few years. Compiler writers and other software producers are rushing to release object-oriented versions of their products. Students strive to be able somehow to list " experience in object-oriented programming" also.

What is object-oriented programming? Why is it so popular? These questions are so popular in our days and there are many answers.

Let's answer the second question first. It is likely that the popularity of object-oriented programming stems in part from the hope, as was the hope for many previous innovations in computer software development, that this new technique will be the key to increased productivity, improved reliability, fewer cavities, and a reduction in the national debt. Although it is true that there are many benefits to using object-oriented programming techniques it is also true that programming a computer is still one of the most difficult tasks ever undertaken by human kind. ( Budd, 1991, pp. 3-5)

The answer for the first question is very difficult because there are many theories and definitions for object oriented programming. All these definitions agree to a common truth, that object-oriented programming is not simply a few features added to programming languages. Rather, it is a new way of thinking about process of decomposition problems and developing programming solutions. Moreover, the user is able to build new abstractions from existing ones by inheriting their properties ( inheritance ), has the ability to identify a type at run time and to manipulate values of several specific types ( polymorphism ), and has the strength to choose an operation at run time ( late binding ) .

Decomposition and inheritance capabilities are provided by Ada 95 through tagged types and child library units, while polymorphism is provided by class wide types, and late binding by dispatching in which the choice of subprogram call is made at run time depending on the type of the parameters or possibly the type of the result of the subprogram call; late binding also occurs through a new form of access type which can reference subprograms.( Ada 9x Mapping / Revision Team, 1994, pp. 4-2)

### 1. Programming By Extension

The key idea of programming by extension is the ability to declare a new type that refines an existing parent type by inheriting, modifying or adding to both the existing components and the operations of the parent type. A major goal is the reuse of existing reliable software without the need for recompilation and retesting.

Type extension in Ada 95 builds upon the existing Ada 83 concept of a derived type. In Ada 83, a derived type inherits the operations of its parent and can add new operations; however, it is not possible to add new components to the type. This static mechanism in Ada 83 has changed in Ada 95 and a derived type can also be extended by adding new components. ( Ichbiah, 1992, pp. 3).

Ada 95 allows only record types to be extended on derivation provided that they are marked as tagged. Private types implemented as records can also be tagged.

Few people are comfortable with the keyword " **tagged** ". The origin of the discomfort is to have a mechanism keyword designate a high-level abstract concept. Moreover, most other major object-oriented languages call these entities classes ( Simula, Eiffel, C++) and the concept is known in the literature as that of class. ( Ichbiah, 1992, pp. 1-2)

An example how Ada 95 implements tagged type ( classes ) as records.

```
type Officer is tagged
    record
        Name  : String   ;
        Rank  : Natural ;
    end record ;
```

Then we can declare

```
type Weapon_Officer is new Officer with
        record
                Subordinate_Officers : Group_of_Officers;
                Subordinate_Enlisted  : Group_of_Enlisted;
                Main_Responsibility_Area : Type_Of_Arms;
        end record;
```

When we have already declared

```
type Group_of_Officers is array ( Positive range < >) of String;
type Group_of_Enlisted is array ( Positive range < >) of String;
type Type_Of_Arms is ( Guns, Surface_Missiles, Air_Missiles, Weapon_Radar);
```

The type Weapon_Officer has five components the Name, Rank, Subordinate_Officers, Subordinate_Enlisted, Main_Responsibility_Area. In this example the type Officer is the parent type ( called the base class or superclass in other object-oriented languages) while the type Weapon_Officer is derived from Officer and is called a child type.

The following example shows a private type that can also be marked as tagged.

```
type Officer is tagged private;
```

and the full declaration must then ( ultimately ) be a tagged record

```
private
        type Officer is tagged
                record
                        Name  : String ;
                        Rank    : String ;
                end record;
```

or the private tagged type can be declared as

```
private
        type Weapon_Officer is new Officer with
                record
                        Subordinate_Officers : Group_of_Officers;
                        Subordinate_Enlisted  : Group_of_Enlisted;
        Main_Responsibility_Area : Type_Of_Arms;
    end record;
```

Just as in Ada 83, derived types inherit the operations which " belong " to the parent type. These are called primitive operations in Ada 95. User-written subprograms are classed as primitive operations if they are declared in the same package specification as the type and have the type as parameter or result.

## 2. Class Wide Programming

The ability to identify a type at runtime and to manipulate values of several specific types is called polymorphism. In Ada 95 polymorphism is provided by class wide types. We need class wide types as mean to manipulate any kind of the parent type and to process it accordingly. ( Skansholm, 1994, pp. 588)

Each tagged type T has an associated type denoted by T'Class. This type comprises the union of all the types in the tree of derived types rooted at T. The values of T'Class are thus the values of T and all its derived types. Moreover a value of any type derived from T can be implicitly converted to the type T'Class.

In the case of Officer example the tree of types can be pictured as in Figure 1.

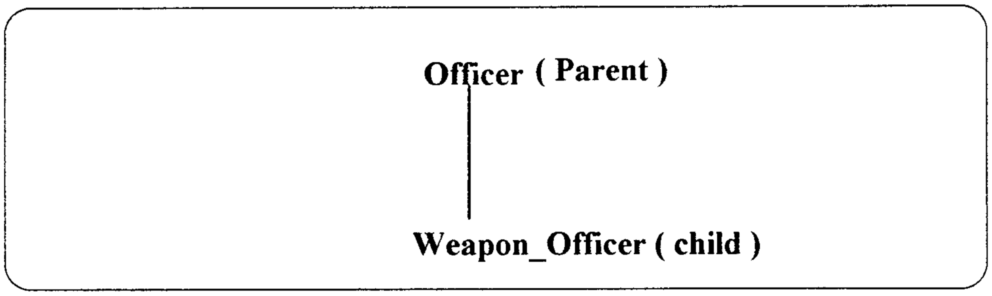


Figure 1 : Tree of Types

A value of any of the Officer types can be implicitly converted to Officer'Class. The Weapon_Officer is not the same as Officer'Class; The Officer'Class consists of the Officer and Weapon_Officer types and their operations when the Weapon_Officer consists just of the Weapon_Officer type and its operations.

Each value of a class-wide type has a tag which identifies its particular type from other types in the tree of types at runtime. It is the existence of this tag which gives rise to the term tagged types.

The type T'Class is treated as an unconstrained type; this is because we cannot possibly know how much space could be required by any value of a class-wide type because the type might be extended. As a consequence, although we can declare an object of class-wide type we must initialize it and it is then constrained by the tag.

***a. Dispatching***

A type T'Class can be used as a parameter of a procedure as in the following example:

```
procedure Process_Officer ( OF : in out Officer'Class) is
      . . .
      begin
      . . .
         Handle_Officer (OF);
      . . .
      end Process_Officer;
```

Procedure Process_Officer takes a class-wide value as its parameter. Procedure Handle_Officer can manipulate any kind of Officer. In this case there are two overloaded versions of the procedure Handle_Officer, one of each type in the tree and so we do not know which procedure Handle_Officer to call until runtime because we do not know which specific type the Officer belongs to ( Weapon_Officer , or Officer ). However, OF is of a class-wide type and so its value includes a tag indicating the specific type of the value. The choice of Handle_Officer is then determined by the value of this tag; the parameter is then implicitly converted to the appropriate specific Officer type before being passed to the appropriate procedure Handle_Officer.

This runtime choice of procedure is called dispatching and is the key to the flexibility of class-wide programming.

### *b. Class Wide Access Type*

A type T'Class can have be referred by an access type. In this case the access could designate any value of the class-wide type from time to time. The following declaration is an example of an access type Officer_Ptr that refers to the Officer a class-wide type :

```
type Officer_Ptr is access Officer'Class ;
```

## 3. Abstract Types And Subprograms

The purpose of an abstract type is to provide a common foundation upon which useful types can be built by derivation. An abstract subprogram is a sort of place holder for an operation to be provided( it does not have a body ).

A tagged type which has one or more abstract operations is said to be an abstract type. An abstract type on its own is of little use because we cannot declare an object of the type.

Upon derivation from an abstract type we can provide actual subprograms for all the abstract subprograms of the parent type. Once all the abstract subprograms are provided, the type is then no longer abstract and we can then declare objects of the type in the usual way. ( Barnes, 1994, pp. 424)

In the example of processing Officer we could reformulate this so that the root type Officer was just an abstract type and then build the specific types upon this. This would enable us to program and compile all the general infrastructures routines for processing all officers such as Process_Officer procedure with out any concern at all for the individual alerts and indeed before deciding what they should contain. The baseline package could then simply become :

```
package Base_Officer_System is
      type Officer is abstract tagged null record;
      procedure Handle_Officer ( OF : in out Officer ) is abstract;
end   Base_Officer_System ;
```

In the Base_Officer_System package we declare the type Officer as a tagged null record with just the procedure Handle_Officer as an abstract subprogram ( This package does not have body).

We can now develop an officer infrastructure and then later add the normal officer system containing the tree types of officers like the following example :

```
with Base_Officer_System ;
package Normal_Officer_System is
          type Executive _Officer is new Base_Officer_System.Officer with
                    record
                              Name: String  ;
                              rank   : Natural;
                    end record;
procedure Handle_Officer( OF : in out Executive_Officer)
type Weapon_Officer is new Executive_Officer with
          record
                    Subordinate_Officers : Group_of_Officers;
                    Subordinate_Enlisted  : Group_of_Enlisted;
                    Main_Responsibility_Area : Type_Of_Arms;
          end record;
procedure Handle_Officer( OF : in out Weapon_Officer);
end Normal_Officer_System ;
```

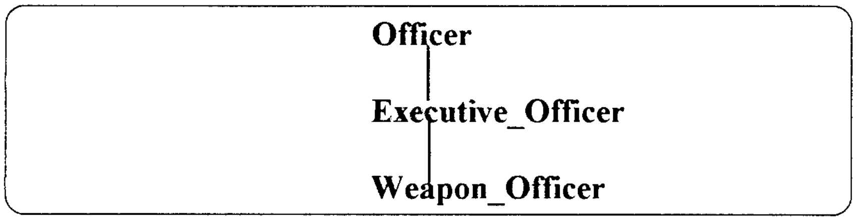


Figure 2 : New Tree Structure

## 4. Late Binding

The ability to choose an operation at runtime is called late binding because the choice of operation is made late in the compile-link-run process. We mentioned dispatching in Ada 95 which is one mechanism for late binding. The second mechanism is provided by the manipulation of subprogram values through an extension of access types.

In Ada 95 an access type can refer to a subprogram; an access-to-subprogram value can be created by the Access attribute and a subprogram can be called indirectly by dereferencing such an access value. The following example demonstrate such a case.

```
type  Trig_Function is access function( F : Float) return Float;
         T        : Trig_Function ;
       X, Theta :  Float;
```

In this example T " point to " functions such as Sin, Cos and Tan. We can assign an appropriate access to subprogram value to T by

```
T := Sin'Access;
```

and later indirectly call the subprogram currently referred to by T as expected

X := T ( Theta); ( Which is really an abbreviation for X := T.all (Theta); )

The access to subprogram mechanism can be used to program general dynamic selection and to pass subprograms as parameters. It allows program call-back to be implemented in a natural and efficient manner. ( Ada 9x Mapping / Revision Team Rationale, 1994, pp. 4- 29)

## B. HIERARCHICAL LIBRARIES

There are occasions when we wish to write two logically distinct packages which nevertheless share a private type. We could not do this in Ada 83. We either had to make the type not private so that both packages could see it with the unfortunate consequence that all the client packages could also see the type; this broke the abstraction. On the other hand, if we wished to keep the abstraction, then we had to merge the two packages together and this resulted in a large monolithic package with increased recompilation costs. ( Ada 9x Mapping / Revision Team Rationale, 1994, pp. 10 - 1 )

Another aspect of the difficulty in Ada 83 arose when we wished to extend an existing system by adding more facilities to it. If we added a package specification then naturally we have to recompile it but moreover we also have to recompile all existing clients even if the additions have no impact upon them.

In Ada 95 these and other similar problems are solved by the introduction of a hierarchical library structure containing child packages and child subprograms.

## 1. Child Units

The Officers process is described by the following package ;

```
package Process_Officer is
        type Officer is private;
        procedure Handle_Officer ( OF : in out Officer);
        private
        type Officer is
              record
                    Name: String  ;
                    rank   : Natural;
              end record;
end  Process_Officer ;
```

In this example sometime later we want to move officers from one department to another by using a new procedure Move_Officers. In Ada 83 we could do this by adding the Move_Officers procedure to the package Process_Officer and this forced us to recompile all the existing clients.

In Ada 95, however, we can add a child package as the following one:

```
package Process_Officer.Move  is
        procedure Move_Officers ( OF1, OF2 : in out Officer) ;
end  Process_Officer.Move ;
```

within the body of this package we can access the private type Officer itself.

The notation, a package having the name P.Q ( Process_Officer.Move ) is a child package of its parent package P. We can think of the child package as being declared inside the declarative region of its parent but after the end of the specification of its parent; most of the visibility rules stem from this model. The child package P.Q solves both the problem of sharing a private type over several compilation units and the problem of existing a package without recompiling the clients. They thus provide another form of programming by extension.( Barnes, 1994, pp. 432)

### 2. Private Child Units

The child units we introduced in the previous section were based around the provision of additional facilities for the client. The specifications of the additional packages were all visible to the client.

In the development of large subsystems it often happens that we would like to decompose the system for implementation reasons but without giving any additional visibility to clients.

Ada 83 had a problem in this area. In Ada 83 the only means at our disposal for the decomposition of a body was a subunit. However, although a subunit could be recompiled without affecting other subunits at the same level, any change to the top level body ( which includes the stubs of the subunits) required all subunits to be recompiled. Ada 95 also solves this problem by the provision of a form of child unit that is totally private to its parent. The example of such a private child is

```
private package P.Q is
    . . .
end P.Q ;
```

The private child P.Q can be declared at any point in the child hierarchy. The visibility rules for private children are similar to those for public child but there are two extra rules.

The first rule is that a private child is only visible within the subtree of the hierarchy whose root is its parent. In addition within that tree it is not visible to the specifications of any public siblings.

The second rule is that the visible part of the private child can access the private part of its parent. This is quite safe because it cannot export information about a private type to a client because it is not itself visible. Nor can it export information indirectly via its public siblings because it is not visible to the visible parts of their specifications but only to their private parts and bodies. ( Barnes, 1994, pp. 438)

## C. PROTECTED TYPES

A protected type has a distinct specification and body in a similar style to a package or task. The specification provides the access protocol and the body the implementation details.

The specification of a protected type is also split into a visible part and a private part. The visible part contains specifications of subprograms and entries providing the protocol. The private part contains the hidden shared data and also the specifications of any other subprograms and entries which are private to the type. ( Ada 9x Mapping / Revision Team Reference Manual, 1994, pp. 165 )

```
protected Variable is
      function Read return Item ;
      procedure Write ( New_Value : Item ) ;
private
      Data : Item;
end Variable ;
protected body Variable is
      function Read return Item is
            begin
                  return Data;
            end Read;
      procedure Write ( New_Value : Item ) is
            begin
                  Data := New_Value;
            end Write ;
end Variable ;
```

The protected object Variable provides controlled access to the private variable Data of some type Item. The function Read enables us to read the current value whereas the procedure Write enables us to update the value. Calls use the familiar dotted notation.

```
X := Variable.Read;
. . .
Variable.Write ( New_Value => Y );
```

Within a protected body we can have number of subprograms and the implementation is such that calls of the subprograms are mutually exclusive and thus cannot interfere with each other. A procedure in a protected body can access the private data in an arbitrary manner whereas a function is only allowed read access to the private data. The implementation is consequently permitted to perform the useful optimization of allowing multiple calls of functions at the same time. ( Ada 9x Mapping / Revision Team Rationale, 1994, pp. 9-4)

## D. EXTENSIONS TO GENERICS

The generic facility in Ada 83 has proved very useful for developing reusable software particularly with regard to its type parameterization capability. However, there were a few anomalies which have been rectified in Ada 95. In addition a number of further parameter models have been added to match the object-oriented facilities and the additional numeric types as described in previous sections.

As a simple example consider the following :

```
generic
    type T is private;
package P is
    . . .
package body P is
    X : T;
    . . .
end P ;
```

In Ada 83 we could instantiate this with a type such as Integer which was fine. However, we could not instatiate this with an unconstrained type such as String because when we came to declare the object T we found that there were no constraints and we could not declare an object as an unconstrained array.

The above problem is solved in Ada 95. The parameter matching rules for the example above no longer accept an unconstrained type such as String but require a type such as Integer or a constrained type or a record type with default discriminants.

While we wish to write a generic package that will indeed accept an unconstrained type we have a new form of notation as follows :

```
generic
        type T ( <>) is private; or
        type T is tagged private; or
        type T is new S; or
        type T is new S with private;
package P . . .
```

In this case we are not allowed to declare an object of type T in the body; we can only use T in ways which do not require a constrained type. The actual parameter can now be any unconstrained type such as string; it could also be a constrained type. ( Barnes, 1994, pp. 446)

## E. EXTENSIONS TO EXCEPTIONS

In Ada 83 exceptions occur when errors or other exceptional situations arise during program execution. An exception represents a kind of exceptional situation, an occurrence of such a situation ( at runtime ) is called exception occurrence. To raise an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called handling the exception.

In Ada 95 we are allowed to mention the same exception more than once in the same handler.

Ada 95 also provides additional facilities which enable a program to identify further information about the cause of an exception. This is particularly useful in an other clause where we may require to log the details of all exceptions raised.

```
        when others =>
                Put ( " Something is wrong ");
        Clean_Up;
    raise;
    end;
```

The above example cannot provide more specific diagnostic information when we use Ada 83 without writing a handler for every exception in the program and, of course, some of these may not be in scope anyway. Ada 95 introduces the idea of an exception occurrence which identifies both the exception and the instance of its being raised.

In Ada 95 Exception_Occurence, Exception_Name, Exception_Message and Exception_Information are declared in the package System.Exceptions. These functions take an exception occurrence as their single parameter and return a string.

By using the functions from the System.Exceptions package we can modify the above exception as the following :

```
when Event : others =>
    Put ( " Unexpected exception : " );
    New_Line;
    Put ( Exception_Message ( Event ) );
    Clean_Up;
    raise;
end;
```

The object Event of the type Exception_Occurence acts as a sort of marker which enables us to identify the current occurrence; its scope is the handler. Such a choice parameter can be placed in any handler. ( Ada 9x Mapping / Revision Team Rationale, 1994, pp. 11-2)

# III. UPWARD INCOMPATIBILITIES

The major design goal of Ada 95 is to avoid or at least minimize the need for modifying the existing base of Ada 83 software to make it compatible with Ada 95. This involves not only pursuing upward compatibility but also preserving implementation dependent behavior that can currently be relied upon.

However, there are some upward incompatibilities imply that some legal Ada 83 programs are no longer legal in Ada 95. The new Ada compiler is guaranteed to identify and reject all such incompatible situations, safeguarding users against unrecognized changes in program behavior. The five most common upward incompatibilities are considered likely to occur in normal programs are ( Ada 9x Mapping / Revision Team Rationale, pp. A-1):

1. New reserved words; in Ada 95, six new reserved words have been added to the language
2. Type Wide Character has 256 positions
3. Unconstrained generic parameters; in Ada 95, different syntax must be used in a generic formal parameter to allow unconstrained actual parameters.
4. Library package bodies illegal if not required; in Ada 95, it is illegal to provide a body for a library package that does not require one.
5. Numeric_Error renames Constraint_Error; in Ada 95, the Numeric_Error has been changed to a renaming of Constraint_Error.

To avoid these five incompatibilities the following guidelines for Ada 83 programs are:

1. The Ada 83 users should not use the reserved words as identifiers:
2. They can not declare the identifiers Wide_String and Wide_Character in package specifications.
3. Do not apply use clauses to package System.
4. Do not declare the identifier Append_File in a package specification. Alternatively, do not apply use clauses to package Test_Io and instantiations of Sequential_Io.

5. For type Character, be prepared that the enumeration will comprise 256, rather than 128, literals. Similarly, for type File_Mode, be prepared for the added literal Append_File.

6. Do not use accuracy constraints in subtype declarations.

7. Put representation clauses for real types immediately after the type declaration.

8. Do not derive from a type declared in the same package. Or if you do, derive the new types before redefining any predefined operations on the parent type.

9. Add a distinctive comment to all generic formal private types that can be legally instantiated with unconstrained types.

10. Do not assume " too much" about the state of the computation when exceptions are implicitly raised. Do not cause implicit exceptions knowingly. Be prepared for the elimination of the exception Numeric_Error.

11. Be prepared for attribute values of real types to more closely reflect the actual hardware. Be aware that accuracy requirements for operations combining fixed-point types with differently base 'Small may be lessened. (Ploedereder, 1992, pp. 1-5)

## A. UPWARD INCOMPATIBILITIES

### 1. Reserved Words

The following reserved words in Ada 95 are not reserved in Ada 83.

#### *a. Abstract*

There are abstract tagged types and abstract subprograms. The purpose of abstract is to provide a common foundation upon which useful types can be built by derivation.

#### *b. Aliased*

Aliased views are the ones that can be designated by an access value.

#### *c. Protected*

The protected types encapsulate and provide synchronized access to the private data of objects of the type without the introduction of an additional task.

*d. Requeue*

A requeue statement can be used to complete an accept-statement or entry_body ( both are used to define potentially queued operations on tasks and protected types), while redirecting the corresponding entry call to a new ( or the same ) entry queue.

*e. Tagged*

Record types can be extended to declare a new type that refines an existing parent type by inheriting, modifying or adding to both existing components and the operations of the parent type.

*f. Until*

**2. Type Character**

In Ada 95, the type Character has 256 positions while in Ada 83, it had 128 positions. As a consequence, we can define characters from other non-English languages like Latin, German, Greek. ( Skansholm, 1994, pp. 77 )

Also an Ada 83 program could be an illegal Ada 95 program if it has a case statement or an array indexed by Character, but it could be a legal Ada 95 program with different semantics if it relies on the position number or value of Character'Last. ( Ada 9x Mapping / Revision Changes from Ada 83 to Ada 9X,, pp. 11)

**3. Library Package Bodies**

In Ada 95, library unit packages are allowed to have a body only if required by languages rules. This avoids a nasty and not so rare error.

In Ada 83, a body need only be provided for a package that really needed one, such as where the specification contains subprogram or task declaration. If a body was provided for a library package that did not need a body, then if the package specification was subsequently changed, the body became obsolete. However, since it was optional, subsequent builds incorporating the package would not incorporate the body, unless it was manually recompiled. This obviously affects packages, for example, that only declare types, constants and/or exceptions, a very common occurrence.( Ada 9x Mapping / Revision Team Rationale, 1994, pp. A-3 )

The solution adopted in Ada 95 is to allow a body for a library unit package only when one is required by some language rule.

### 4. Indefinite Generic Parameters

In Ada 95, additional syntax is needed to indicate that a generic actual type is allowed to be indefinite. Otherwise, the actual must be definite.

In Ada 83, no indication was given in a generic formal type declaration as to whether the actual needed to be definite, for example because the body declared an uninitialized variable for the type. It was thus possible for a legal instantiation to become illegal if the body was changed.

An Ada 83 program, where an indefinite type is used as a generic actual parameter is an illegal Ada 95 program.

### 5. Numeric Error

In Ada 95, the exception Numeric_Error is declared in the package Standard as a renaming of Constraint_Error. The checks that could cause Numeric_Error to be raised in Ada 83 have been reworded to cause Constraint_Error to be raised instead.

## B. GUIDELINES FOR ADA 83 PROGRAMS

### 1. Do Not Use The Reserved Words

These reserved keywords have been added to Ada 95. Clearly, programs containing such identifiers will be rejected as syntactically illegal by Ada 95 compiler.

The following declarations show such incompatibilities.

```
Protected : Boolean := True ;
procedure Requeue ( The_Activity : Activity; On_Queue : Queue ) ;
```

These declarations are legal in Ada 83 but illegal in Ada 95 because 'protected ' and ' requeue' are reserved words. Detection of that incompatibility is straightforward but automatic correction is problematic; to ensure that name change is valid requires significant analysis especially if the identifier is the name of a library unit, or occurs in a package specification for which use clauses occur.

### 2. Do Not Declare The Identifiers Wide_String, Wide_Character

Do not declare the identifiers Wide_String and Wide_Character in package specifications because these identifiers have been added to package Standard of Ada 95. As a consequence, user-defined names with the same identifier, when made directly visible via use-clauses, will no longer denote the imported definitions but those in package Standard. The rules of Ada give precedence to directly visible names over names made directly visible by use-clauses. Theoretically, the program will still be legal, but will behave differently in execution. In practice, however, it is likely that program illegalities will occur as a consequence of the altered binding. For example the declaration

```
Str : Wide_String ;
```

in Ada 83 denotes PKG.Wide_String while in Ada 95 denotes the package Standard.Wide_String.

A secondary consequence of the appearance of the new types Wide_Character and Wide_String for the support of international character sets is that operations involving only character or string literals may become ambiguous, since the operation could be for either the Ada 83 types or the new Ada 95 types. For example, the equalities in

```
if 'a' = 'b' then . . .
if "abc" = "def" then . . .
```

are now ambiguous. The same applies if the operands are overloaded function calls that cannot be disambiguated by their parameters. Given that no competent programmer would compare two string or character literals, this situation is not worth a guideline. However, preprocessors might generate such code. The solution is to type-qualify one of the arguments of the expression. The appearance of a variable anywhere in the expression avoids the ambiguity.

### 3. Do Not Apply Use-Clauses To Package System

This is generally a good Ada 83 guideline since implementors are free to add additional declarations to package System. Consequently, a new release of a compiler may declare new names in the package, causing visibility conflicts or ambiguities in code that already makes equally named declarations from other packages directly visible via use-clauses. In Ada 95 adds additional predefined names in package System, and hence can cause this problem. Unlike the situation discussed in the previous paragraph, encountering this incompatibility will make the Ada 83 program illegal in Ada 95, since the Ada visibility rules will then make the conflicting, non overloadable declarations from neither package directly visible.

### 4. Do Not Declare The Identifier Append_File

Ada 95 adds the enumeration literal Append_File to the File_Mode type in these packages. As an additional name in a predefined package, this can create the problems already described under the previous guideline. Alternatively, do not apply use clauses to package Text_Io and instantiations of Sequential_Io.

### 5. Enumeration Type Incompatibilities

For type character, be prepared that enumeration will comprise 256 rather than 128, literals. Similarly, for type File_Mode be prepared for added literal Append_File.

These changes have two major consequences :

The 'Last attribute applied to type Character ( or any of its derived types ) will yield a value with an encoding of 255 rather than 127. The example

```
if ASCII.DEL = Character'Last then
```

is always true in Ada 83 but false in Ada 95. Similarly File_Mode'Last will now yield Append_File. Whether or not this impacts an algorithm depends entirely on the programmer's assumptions. If the assumption was to obtain truly the last character in the enumeration, regardless of its encoding, the code will still operate as intended. If the specific value is important, the explicit literal ( or a declared constant ) should be used instead of the attribute. An analogous reasoning applies to the 'Range attribute applied to arrays with such

index types or to any implicit assumptions about the magnitude of the enumeration. The following example shows how Ada 83 code will not be upward compatible.
The loop will result in a Constraint_Error in Ada 95, beside being poor style in Ada 83.

```
for CHR in Character'First .. Character'Last  loop
        BAD_TT(Character'Pos(CHR)) := . . . ;
end loop;
```

Case statements over expressions of such types need to account for the possible choices added by Ada 95. Otherwise, the statements will be illegal in Ada 95. Since superfluous others or an empty range choice is legal in Ada, case statements, whose choices do not already cover all values permitted by Ada 95 should be written with such an added choice. Alternatively, one can also qualify the case expression with a static subtype, reflecting the Ada 83 range of types Character and File_Mode respectively. The following example illustrates this situation.

```
case CHAR is
  when Character'Val (0) .. Character'Val( 63)
      => . . .
  when  Character'Val (64) .. Character'Val( 127)
      => . . .
end case;
```

This case statement is illegal in Ada 95 since not all choices are covered ( from 1 to 256 ). A solution in Ada 95 is by using the form

```
when others => null;
```

Hence the correct case statement will be :

```
case CHAR is
  when Character'Val (0) .. Character'Val( 63)
      => . . .
  when  Character'Val (64) .. Character'Val( 127)
      => . . .
```

```
 when others => null;
end case;
```

## 6. Do Not Use Accuracy Constraints In Subtype Declarations

Accuracy constraints in subtype declarations will be removed from the language. It is surmised that all existing Ada implementations simply ignore such subtype constraints anyhow, computing and storing results in the accuracy of the respective type.

In existing programs, such accuracy constraints should be removed. They can be found via editor scripts or syntactic searches. The following example

```
type VOLT is delta 0.125 range 0.0 .. 255.0;
subtype ROGH_VOLTAGE is VOLT delta 1.0 range 0.0 .. 100.0;
```

should be replaced with

```
type VOLT is delta 0.125 range 0.0 .. 255.0;
subtype ROGH_VOLTAGE is VOLT range 0.0 .. 100.0;
```

## 7. Representation Clauses For Real Types

Ada 95 solves an obscure problem in Ada 83 by mandating that such representation clauses be given before any subtypes or derived types of the given type are declared.

In existing the programs, once the situation is identified, the correction is very simple; merely move the representation clause to precede the subtype or derived type declarations.

## 8. Derived Types

Do not derive from a type declared in the same package or if you do, derive the new types before redefining any predefined operations on the parent type. In Ada 83, subprograms become derivable only at the end of the visible part of the package in which the type is declared. In Ada 95, they are immediately derivable. As shown in the following example,

```
package Derivations is
     type Byte is array ( 0..7) of Boolean;
     function "and" ( L, R : Byte ) return Byte; -- redefine "and" for whatever reason
```

```
type CHAR is new Byte;
-- In Ada 83, this type inherits the predefined "and"
-- In Ada 95, this type inherits the redefined "and" of Byte
```

a sequence of declarations, interspersing redefinition's of predefined operations with derived type declarations yields different definitions of these operations for the derived type. The guideline avoids this situation, which has surprised many first-time Ada users.

In existing programs, these situations can best be identified by searching for redefinitions of predefined operations, which should be relatively rare, and inspecting the context of these redefinitions for subsequent derived type declarations. If such situations are found, a simple recording of the declarations makes the programs upward compatible.

## 9. Generic Formal Private Types

Add a distinctive comment to all general formal private types that can be legally instantiated with unconstrained types. In Ada 83, the instantiations of a generic with an unconstrained type can be illegal, depending on the use of the type in the generic unit, e.g., in an object declaration. In Ada 95, this instantiation will always be illegal, unless the formal type is identified as one that can be instantiated with an unconstrained type ( in which case, the instantiation will always be legal and the generic must not contain conflicting uses of the type). Ada 95 uses the box notation ( <> ) for this purpose. The example shows this situation.

```
generic
      type X is private;  -- Ada 95 : add (<>)
procedure Ada_83;
type  Short_String is new String (1..8);
procedure Good_One is new Ada_83 ( Short_String );
procedure Dubious_One is new Ada_83 ( String);  -- (1)
-- may be illegal; in Ada 83, depending on generic body of Ada_83. It will always be
-- illegal in Ada 95

generic
```

```
        type X (<>) is private;  -- Ada 95 notation; illegal in Ada 83
    procedure Ada_94;
    procedure Better_One is new Ada_94 ( String);  -- always legal
```

The legality of the instantiation (1) in the above example will depend on the use of type X in the body of 'Ada_83". This is the single instance where Ada 83 code cannot be written to be upward compatible with Ada 95. Consequently, we recommend the use of a distinctive comment convention, as shown at the beginning of the example, so that upon transition to Ada 95, the necessary source changes are easily located and made.

For existing programs, there is a mechanical method of establishing whether or not such a comment should be added: identify all generic units with formal private types and instantiate them with unconstrained types. All such instantiations not rejected by the Ada 83 compilation system ( possibly at link time ) are candidates to be commented as indicated.

**10.Implicit Exceptions**

Do not assume "Too Much" about the state of the computation when exceptions are implicitly raised. Do not cause implicit exceptions knowingly. Be prepared for the elimination of the exception Numeric_Error. Software engineers generally agree that knowingly utilizing implicitly raised exceptions as a control flow mechanism is a highly dubious practice. The interpretation of Ada 83 regarding the execution order in the presence of implicitly raised exceptions is far from clear. Efficient support of RISC, pipelined, super-scalar, and massively parallel architectures makes it necessary to deviate from a purely sequential execution model. Ada 95 is likely to better codify permission for such support in compilers.

Consequently, users should not rely in their algorithms on the specific point in the sequential execution order where an implicit exception may get raised. A good, although over-conservative, guideline is to assume that, in any exception frame, the raising of implicit exceptions may occur as early as the beginning of the frame or as late as the end of the frame, regardless of where the exception occurs in the sequential execution order.

This guideline already applies to Ada 83 code, regardless of Ada 95. Yet, it will avoid additional surprises when upgrading the code to Ada 95.

In some situations, where Ada 83 mandates the raising of Constraint_Error, the semantics of Ada 95 will cause the "expected" result, rather than an exception. Since all these cases are at fringes of the language usage, and since programmers should not cause Constraint_Error intentionally, they are not worth a guideline.

Ada 95 is very likely to eliminate the exception Numeric_Error from the language definition. An official Ada 83 interpretation advises nonbindingly, that Ada 83 implementations should use Constraint_Error instead of Numeric_Error.

### 11. Attribute Values Of Real Types

Be prepared for attribute values of real types to more closely reflect the actual type representation. Be aware that accuracy requirements for operations combining fixed-point type with differently base "Small May Be Lessened". For the casual user of real types, these language changes are most unlikely to have any impact. In particular, users of the predefined floating-point types should have no problems in upgrading to Ada 95. For numeric applications that parameterize algorithms by means of real type attributes, however, the changes to attribute values may have some effect, albeit a positive one, since such applications are presumably interested in more accurate attribute values. ( For example, it is proposed that the attribute 'SMALL return the actual smallest representation increment, rather than the increment between model numbers). Such code is typically written by numerical analysts who, as a group, have requested these changes be made to Ada 83.

The reduced accuracy requirements for operations combining fixed-point types with differently based scales is unlikely to have any significant impact because such operations are exceedingly rare. Also, many Ada 83 compilers support only binary and decimal 'Smalls, thereby making it rather unlikely that existing code applies such operations; they rely on the accuracy required in Ada 83.

# IV. COURSE OUTLINE

## A. WHY USE ADA 95 AS THE BASIC PROGRAMMING LANGUAGE

### 1. Modern Programming Language

We mentioned in previous chapters the Ada 95 strong capabilities like, object oriented design, modularity, information hiding, large scale programming, interfacing to other systems. The above features are considered indispensable by modern software engineering and provide expressive power, maturity, and softwarebase. In addition, Ada is superior to other general programming languages ( C++, Pascal, Eiffel) in terms of embedded, real-time, and security features. The comparison of programming languages is in part a subjective affair; judgments are influenced by personal stylistic choices, by familiarity, by well known first-language effect, etc. and it is not the scope of this thesis but it is trivial that " anything you can do in any modern programming language, you can do in Ada 95 as well".

### 2. Department Of Navy Policy

The policy implemented by the Department of Navy requires the use of the programming language Ada in the development or maintenance of the software systems. The Naval Postgraduate School as the high level education source must provide the Officers both the knowledge and the training which will be used in the future to help the Navy growth and expand in many scientific areas. Since all the software systems which are used today by the Navy and general by the DoD are written by using Ada as programming language, it is obvious that the Ada must be taught as the basic programming language by Computer Science Department. (SECNAV INSTRUCTION 5234.2A, April 1994)

## B. COURSE CONSTRUCTION

In this section we describe the design and development of new course material for the new Ada. For the reasons we mentioned above Ada 95 will be the basic general programming language for Computer Science Department and thus it expresses the continuation of the traditional Structure Programming with Ada or CS 2970 course.

Furthermore, we decide, by taking in to account the computer science students background, that it is compulsory to start teaching first the core Ada 95 language and later the recently improvements, like the object oriented programming features. We are guided to this decision about the course material structure because it is tremendously difficult for the students to understand the new object oriented programming features of Ada 95 like Programming by Extension, Class-Wide Programming, Abstractions, etc. without having in advance experience on the core language's capabilities like record types, tagged records, abstract types, packages, access types, exceptions, generic packages, etc.

The above reasons lead us to divide the new Ada course material in two parts; (1) the core Ada 95 programming language and (2) the object oriented programming with Ada 95. The sources that provided us the information to build this new course material are the following :

1. Ada 9x Project Office ( access by anonymous ftp from address ajpo.sei.cmu.edu)
2. GNAT compiler ( access by anonymous ftp from address cs.nyu.edu)
3. Ada 9x Reference Manual.
4. Ada 9x Rationale.
5. Ada Problem Solving and Program Design written by Michael B. Feldman & Elliot B. Koffman.
6. Programming in Ada Plus an Overview of Ada 9x written by J.G.P. Barnes.
7. Ada from the Beginning second edition( 1994) written by Jan Skansholm.

## 1. Core Ada 95 Language

In the core language material part we make an introduction to the programming languages and a short history of Ada and we continue by representing the features of new Ada like the Lexical Elements ( Identifiers, Numeric Literal, Character Literal, String Literal), Declaration and Types ( types, subtypes, scalar types, record types, array types, access types), Expressions, Statements ( Assignment, if, case, loop, block, exit statements), Subprograms ( procedures, functions), packages ( package specification, package body, generic packages), Exceptions and we finish with a short presentation of the Abstract Data Types ( ADT ).

The whole course material constructed based on students background, from which we assumed that every student must have experience on at least one other programming language like C, Pascal, Fortran, etc. This core language part was scheduled to last until the ninth week in the quarter, and will contain the first Midterm Exam.

To present each of the core Ada's 95 semantics we constructed the slides which are represented in Appendix A.

## 2. Object Oriented Programming

In the second part we present the Ada's 95 object oriented approach which is accomplished by Programming by Extension, Class Wide Programming, Abstract Types and Subprograms, moreover we present the new features which are offered by new Ada like Discriminants, Operations of Tagged types, Access Types, Hierarchical Libraries, Exceptions and Generic packages.

This second part is scheduled to instruct from ninth to eleventh week in the quarter and will contain the second Midterm Exam.

As in the core language part we built for each semantic of object oriented programming slides which are represented in Appendix B.

# V. CONCLUSION

## A. ACCOMPLISHMENTS

Ada has been designated by the Department of Defense as the mandated language for all the DoD software development. Furthermore, its user base has expanded to include all major defense agencies of the Western world, and increasingly many areas in civil and private sectors. After six years of use and many implementations, the DoD made a decision to undertake a revision to Ada. In 1995, the new Ada or Ada 95( Ada 9x) as named was underway and is a natural extension to the original design of Ada .

### 1. An Overview Of Ada 95 Language

We demonstrated the critical changes of the new Ada by representing the new Object Oriented programming capabilities, Hierarchical Library features, Tasking enhancements, and the new features in Generics packages, Extensions. This introduction to the new Ada facilities presents the near perfect compatibility between Ada 83 and Ada 95. The compatibility which derived from DoD's requirement document, was accomplished in essence by the establishment of a new record type named tagged. In addition, all the other new features are produced based on those tagged record types.

### 2. Incompatibilities

Although Ada 95 is compatible to Ada 83 there is a small number of incompatibilities which have been adopted. The new reserved words, the 256 type character's positions, unconstrained generic parameters, library package bodies, and Numeric_Error renames Constraint_Error are incompatibilities that are considered likely to occur in normal programs. These few incompatibilities can be dealt with in a simple mechanical way.

### 3. Guidelines To Ada 95 Programmers

The few incompatibilities between the revised Ada and Ada 83 arise from language changes to correct Ada 83 problems. Some of the incompatibilities are so specialized that the probability of there occurrence is minimal. We presented some guidelines that refer to all the known incompatibilities, and thus providing Ada 83 code that will be compatible with Ada 9x.

### 4. Course Outline For The Core Language

We built a course outline for the core Ada 95 language based on the Computer Science Department's needs and students background. The way to construct a course outline was by presenting the new Ada's terminology, definitions, and examples for each of the core language's features. At the end of each section we built case studies which we strongly believe that make the student to understand easier the material that had already instructed.

### 5. Course Outline For The Object Oriented Programming With Ada 95

Since, the object oriented programming facilities is a new programming style and moreover, presents the main difference between the Ada 95 and Ada 83 we decided to built a separate course outline for these object oriented programming capabilities. The construction of this material was based on the philosophy followed in the core language.

## B. RECOMMENDATIONS

### 1. Ada 95 The Basic General Programming Language For Computer Science Department

Ada 95 contains all the capabilities like object oriented design, modularity, interfacing with other systems, information hiding, programming in large, etc. which are offered by the modern programming languages. Moreover, the DoD policy requires the use of Ada as a general programming language for all software systems. The analysis of the above features and rules, shows the necessity to teach Ada 95 as a basic programming language for Computer Science Department.

### 2. Structured Programming With Ada 95

We propose the new course to continue having, the same course number CS-2970 and title Structured Programming with Ada 95 for historical reasons. Moreover, the proposed schedule for this course is divided into two parts; the first part will cover the core Ada 95 language, as well as it will be taught up to the ninth week, and the second part ,presents the object oriented programming with Ada 95 and it will be instructed to the last three weeks.

### 3. Textbook

Since the last face of Ada's 95 standardization process has not been competed, few textbooks have been published about Ada 95. The small domain of the published textbooks limit the reference material for students. At this time we propose Programming in Ada plus An Overview Of Ada 9X , a textbook written by J.G.P. Barnes in 1994, which was the appropriate textbook from the already published because it refers extensively to the Ada 95 features and particularly in the object oriented programming facilities. It is hoped this thesis will be a supplementary text for the near future.

### 4. Future Work

The Ada 95 language has not approved yet as an ANSI/ISO standard and thus the research field on this language is huge. There are many research topics in Ada 95 and also a lot of comparison topics between Ada 95 and general programming languages that are implemented today, about the capabilities, effectiveness, feasibility, for the new Ada. For the near future the research on Ada 95 can focus to the new advance course establishment and the comparison between Ada 95 and C++ which is the most implemented general programming language in the market.

#### *a. Advanced Ada 95 Course*

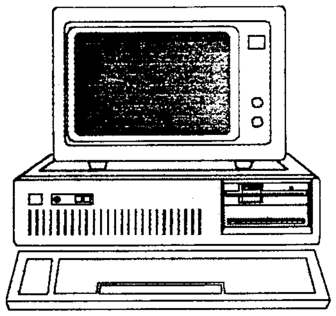
The construction of a new advance Ada 95 course in which the students will have the opportunity to learn the advance object oriented programming, and tasking facilities with the Ada 95.

#### *b. Comparison Ada 95 And C++*

Since C++ is another modern programming language with the same capabilities with Ada 95, it will be a good thesis topic to contrast C++ and Ada 95 .

# APPENDIX A

## CS 2970 Viewgraphs



Computer Science Department

Naval Postgraduate School

Monterey, CA 93943

**CS 2970- Structured Programming in Ada**

- **Objective :** Students completing CS 2970 will be able to discern the functionality of Ada software, develop software from a detailed functional description and employ the features of a subset of the language effectively.

- **Assumptions:** Students have had no systematic introduction to programming.

  Students have interest in programming (at least, to use as problem-solving tool).

- **Course Contents:** Introduction to computers and programming

  Reading programs

  Problem solving for computer implementation

  Ada language elements

- **Course Grading :**

  2 Midterm Exams- 25% each

  5 projects - successfully completed

  Expect to work hard

- **Text:**

  Programming in Ada Plus an Overview of Ada 9x J.G.P. Barnes 1994

**The Criticality of Software**

Hardware is no longer the dominant factor in the hardware / software relationship.

The demand for software is rising exponentially.

The cost of software is rising exponentially.

Software maintenance is the dominant software activity.

Systems are getting more complex.

Life and property are dependent on software.

**Program**

PROGRAM n. A logically arranged set of programming statements ( or instructions ) defining the operations to be performed by a computer in order to achieve the desired results. tr.v. To write a program in order to solve a problem or to control the operation of a computer.

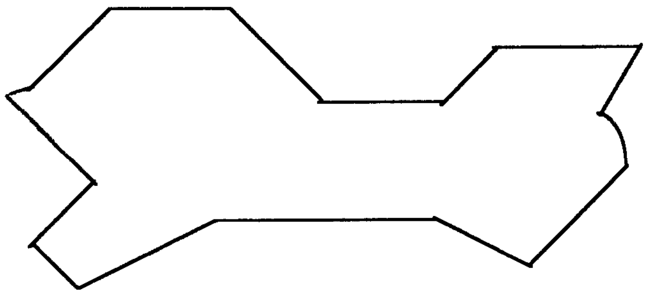
PROGRAM n. A magic spell cast over a computer allowing it to turn one's input into error messages. tr.v. To engage in a pastime similar to banging one's head against a wall, but with fewer opportunities for reward.

**Language Generartions**

**Zero Generation :** (1940 - 1954)

Machine languages, Assemply languages

Instructions and Data



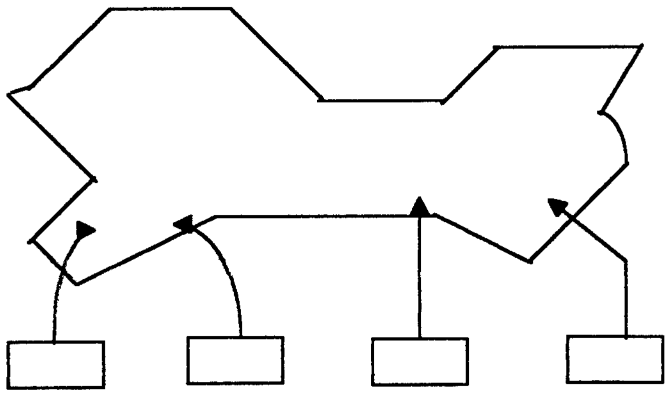
**First Generation :** ( 1954 - 1958 )

FORTRAN-I, ALGOL - 58, IPL V

**Second Generation :** ( 1959 - 1961 )

FORTRAN-II, ALGOL - 60, COBOL, LISP, C ( later)

Data

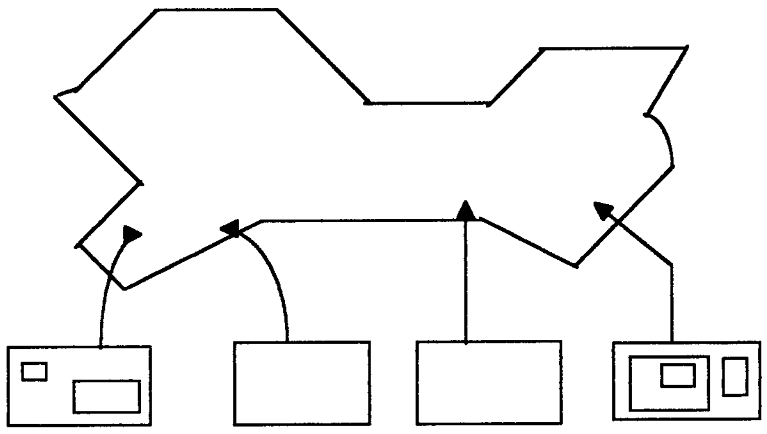


**Subprograms**

**Third Generation :** (1962 - 1970 )

PL/I, ALGOL - 68, Pascal, SIMULA, JOVIAL

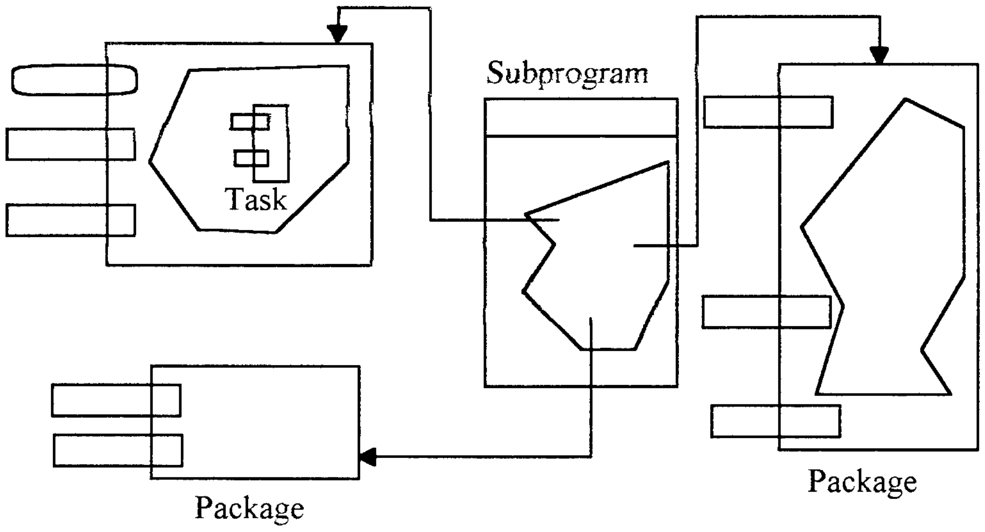
Data



**Subprograms**

**Generation Gap :** ( 1970 - ? )

Modula, C++, Ada



**Why Ada?**

**From 1968-1973, DoD software costs increased 51%**

- 450 general purpose languages for DoD systems
- 500 - 1500 language processors
- Rapidly escalating training and maintenance costs

**56% of costs were for embedded computer software**

- Large ( 1,000 - 1,000,000 SLOC )
- Long-lived ( 10 - 15 year life-span )
- Continuous change
- Subject to physical constraints
- Reliability critical in fault prone environment

**HOLWG founded**

- DoD, representatives from other agencies, liaisons from UK, West Germany, France.
- Develop requirements for languages.
- Evaluate existing languages against requirements.
- Recommend minimal set of languages.

**Why Ada 95 ?**

Ada 83 had not the capabilities to be implemented on :

- Large-scale information systems
- Distributed systems
- Large scientific computation
- Systems programming
- Object Oriented Design

Ada 95 contains all the features to construct complex systems as the above.

**Language Requirements Series**

**Circulate to DoD, Federal Agencies, Industry, Academia, Europe**

- 85 DoD organizations ( principal author at IDA )
- 26 Industrial contractors
- 16 Academic institutions
- 7 other organizations

**April 1975- STRAWMAN**

**August 1975- WOODMAN**

**January 1976- TINMAN**

- Basis for DoD Directive 5000.29, mandating only DoD- approved languages in defense systems.
- Basis for evaluation of existing languages.
- Result: No single existing language suitable for DoD embedded computer systems.

**January 1977- IRONMAN**

- Reformat and correction of TINMAN
- Basis for RFP of DoD Common High-order Language ( then called DoD-1).
- Four contractors selected to develop language designs ( Blue/ Yellow/ Red/ Green).

**June 1978- STEELMAN**

- Green contractor ( Honeywell/ Bull) won design.
- Result language retitled ada

**Augusta Ada Byron**

Augusta Ada Byron, the Countess of Lovelace daughter of Lord Byron, the poet ( 1815 - 1852 ).

The countess was a mathematician and is credited as the world's first programmer in light of her work with Charles Babbage and his Difference and Analytic Engines.

**April 1988**

DoD initiated a revision of ANSI/MIL-STD 1851A ( Ada 83).

**December 1990**

Requirement document underway.

**January 1991**

Process of Mapping .

**January 1993**

First phase approval as an ANSI/ISO standard

**March 1994**

First step of the second phase approval .

**1995**

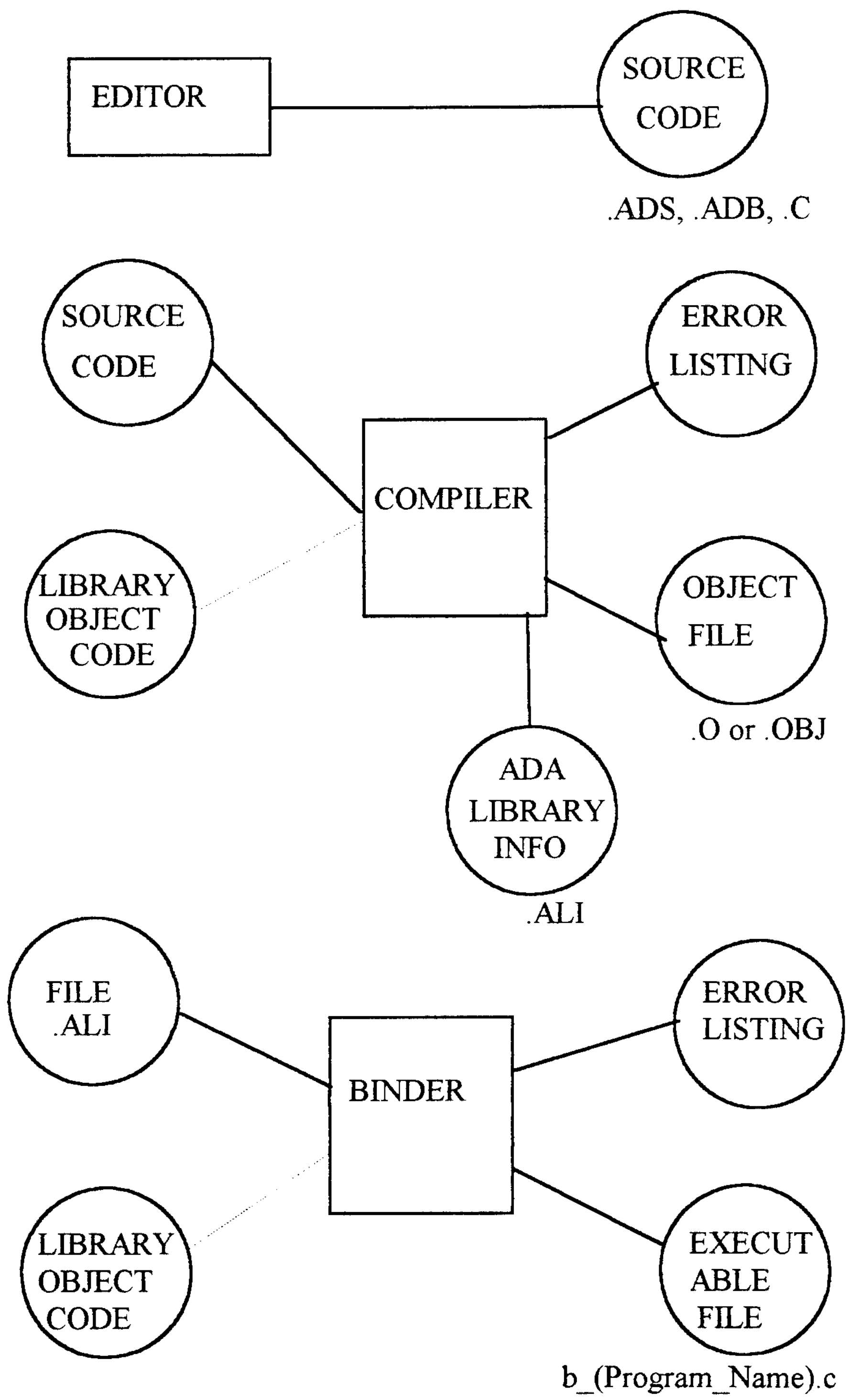
Second phase as an ANSI/ISO standard

**Ada Program Structure**

```
with Package;
use Package;
procedure Program_Name is
    declarations -- defines logical entities to be used in this program
begin
    Statement 1;
    Statement 2;
    Statement 3;
    .
    .
    .
    Statement N;
end Program_Name;
```

**A First Ada Program**

```
-- CS 2970 Example Program 1
-- This is a short program that prints a one-line fixed message consisting of
-- Hello there We hope you enjoy studying Ada 94.
-- You have to name your file hello.ads ( or hello.adb)
with Text_IO;
use Text_IO;
procedure hello is
begin
    Put ( Item => "Hello there ");
    Put ( Item => "We hope you enjoy studying Ada 94:);
end hello;
```


EDITOR
SOURCE CODE
.ADS, .ADB, .C
SOURCE CODE
ERROR LISTING
COMPILER
LIBRARY OBJECT CODE
OBJECT FILE
.O or .OBJ
ADA LIBRARY INFO
.ALI
FILE .ALI
ERROR LISTING
BINDER
LIBRARY OBJECT CODE
EXECUT ABLE FILE
b_(Program_Name).c

**Errors**

➢ Syntax

Rules of the language have been broken

➢ Environment

Assumptions of the support system are not met.

➢ Logic

Algorithm is incorrect.

**English Examples :**

The monkey ate the banana

monkey ate banana

The monkey ate 5 tons of bananas that day.

The banana ate the monkey.

**Ada Examples:** ( First statement, previous page)

```
Put("Hello there.");
Put(Hello there.);
Fut("Hello there.");
Put("Hell here");
```

**Second Example**

```
-- CS 2970 Example Program 2
-- This is a short program that uses variables
-- You have to name your file HelloInitials.ads ( or HelloInitials.adb)

with Text_IO;
procedure HelloInitials is
        Initial1 : Character;
        Initial2 : Character;
begin
        Text_IO.Put (Item => "Enter your two initials> ");
        Text_IO.Get (Item => Initial1);
        Text_IO.Get (Item => Initial2);
        Text_IO.New_Line;
        Text_IO.Put (Item => "Well ");
        Text_IO.Put (Item => Initial1);
        Text_IO.Put (Item => Initial2);
        Text_IO.Put (Item => ".  We hope you enjoy studying Ada!");
        Text_IO.New_Line;
end HelloInitials;
```

**Design Process**

1. State the problem clearly as a series of steps.
2. Describe the input and output of each step.
3. Work a sample problem by hand.
4. Develop an algorithm* to solve the problem.
5. Test the algorithm with a variety of data. Check extreme cases.
6. Express the algorithm as a program in programming language.
7. Compile and run the program on the computer.

**Biggest Time Waster in past classes:**

Starting at step 6

***Algorithm**

* Description of how a particular problem should be solved.

**Top-down design**

1. Divide a problem into subproblems.
2. Solve the subproblems individually.
3. Divide the subproblems into further subproblems.
4. Continue in this way until all the subproblems are easily solvable.

**Ada Names ( Identifiers )**

➢ Consists of a letter possibly followed by one or more letters or digits with embedded isolated underlines.

**2not_Correct_**

➢ All the characters which are interpreted as letters in the ISO standard can be used. Case of the letters does not change the meaning.

➢ Ada does not impose any limit on the number of characters.

➢ Use meaningful names **TimeOfDay** not **T**.

➢ Cannot use any of the 69 reserved words.

➢ Avoid using standard identifiers ( character, integer ).

**Examples:**

Count,

X or (x),

Get_Symbol,

X1 or ( x1),

Snob_4,

Straβe,

CITTÀ

## Reserved Words

| | | | |
|---|---|---|---|
| **abort** | **else** | **new** | **return** |
| **abs** | **elsif** | **not** | **reverse** |
| **abstract** | **end** | **null** | |
| **accept** | **entry** | | **select** |
| **access** | **exception** | | **separate** |
| **aliased** | **exit** | **of** | **subtype** |
| **all** | | **or** | |
| **and** | **for** | **others** | **tagged** |
| **array** | **function** | **out** | **task** |
| **at** | | | **terminate** |
| | **generic** | **package** | **then** |
| **begin** | **goto** | **pragma** | **type** |
| **body** | | **private** | |
| | **if** | **procedure** | |
| **case** | **in** | **protected** | **until** |
| **constant** | **is** | | **use** |
| | | **raise** | |
| **declare** | | **range** | **when** |
| **delay** | **limited** | **record** | **while** |
| **delta** | **loop** | **rem** | **with** |
| **digits** | | **renames** | |
| **do** | **mod** | **reque** | **xor** |

**Ada Values ( Literals )**

An alphanumeric value defined as a constant in a program.

➢ Universal Integer :

12 0 1E6 -17 123_456

➢ Universal Real :

12.0 0.0 0.456 -3.25 3.14158_26 1.34E-12 1.0E+6

➢ Character :

'A' 'a' 'Z' '5' '0' 'Ξ'

➢ String :

"Hello" "Press the key" "A" " " """"

"First part of a sequence of character" & "that continues on the text"

"the letter '?' is strange"

➢ Enumeration

True False Sat Green LCDR

## Ada Data Objects

- ➢ A program manipulates data objects.
- ➢ An object represents something that occurs in the real word.
- ➢ Objects with different properties have different types.

**TYPES characterized by:**

- ➢ Values that can be taken by objects of the type.
- ➢ Operations that can be carried out on objects of the type.

Declarations with Standard Types

```
Number1             : Integer;

Number2,Number3     : Natural;

Class_Number        : constant Integer :=2970;

Answer              : Float;

Cost,
Sales_Tax,
Total_Amount        : Float;

Initial1            : Character;

Name                : String(1..15);

New_Name             : String:="David";
Found               : Boolean;
```

## Input and Output of Numbers

```
-- I/O procedures for standard numbers
-- Precompiled instantiations of Integer_Io and
-- Float_Io for the predefined Integer and Float types

        with Text_IO;
        package My_Int_IO is new Text_IO.Integer_IO (Num => Integer);

        package My_Flt_IO is new Text_IO.Integer_IO (Num => Float);
```

**Number Input/Output Example**

```
-- CS 2970 example 3
-- A small example of input and output of float values in Ada


with Text_IO;
with My_Flt_IO;
procedure InchToCM  is
        CM_Per_Inch : constant Float := 2.54;
        Inches      : Float;
        Centimeters : Float;
begin
        Text_IO.Put (Item => "Enter a length in inches> ");
        My_Flt_IO.Get (Item => Inches);
        Centimeters := CM_Per_Inch * Inches;
        Text_IO.Put (Item => "That equals ");
        My_Flt_IO.Put (Item => Centimeters);
        Text_IO.Put (Item => " centimeters");
        Text_IO.New_Line;
end InchToCM;
```

**Assignment and Expressions**

Assignment :

Variable := Expression;

Type of variable and expression must be identical

A:= 5; -- valid if A is declared Integer or Natural

A:=5.0; -- valid only if A is declared float

A:="Ada"; -- valid only if A is declared String (1..3) or ( e.g. String(7..10))

Expression

Literal values ( 5,3.2, -1, 'a', " Cat", 'Ξ', "Straβe")

Attribute values ( discussed later )

Operators and values

How it works

The value of the Expression on the right-hand side is evaluated first.

This value is placed in the Variable (which appears on the left-hand side)

The previous values of the variable are destroyed.

The Variable gets the new value from the Expression

## Operators

### Precedence of Operators

Parentheses can be used to determine precedence;
otherwise, operators are evaluated as follows ;

| | |
|---|---|
| 1. ** abs not | -- highest precedence |
| 2. * / mod rem | --( multiplying operators) |
| 3. + - | -- ( unary) |
| 4. & + - | -- ( binary adding ) |
| 5. = /= < <= > >= in notin | -- ( relational operators ) |
| 6. and or xor and then or else | -- ( logical operators ) |

An explicit declaration of an operator = is permitted for all types.

**Case Study : Value of a Coin Collection**

**Problem**

A child has been saving nickels and pennies for quite a while. Because she is getting tired of lugging her piggy bank with her whenever she goes to the store, she would like to trade in her collection for dollar bills and some change. In order to do this, she would like to know the value of her coin collection in dollars and cents.

**Analysis**

To solve this problem, we must be given the count of nickels and the count of pennies in the collection. The first step is to determine the total value of the collection in cents. Once we have this figure, we can do an integer division using 100 as the divisor to get the dollar value; the remainder of this division will be the loose change that she should receive. In the data requirements, we list the total value cents (TotalCents) as a program variable because it is needed as part of the computation process; it is not a required problem output.

**Data Requirements**

**Problem inputs :**

Nickels : Natural ( the number of nickels )
Pennies : Natural ( the number of pennies )

**Problem outputs :**

Dollars : Natural ( the number of dollars she should receive )
Change : Natural ( the loose change she should receive )

**Additional program Variables**

TotalCents : Natural ( the total number of cents )

**Relevant Formulas**

One nickel equals five pennies.

**Design**

**Initial Algorithm**

1. Read in the count of nickels and pennies.
2. Compute the total value in cents.
3. Find the value in dollars and loose change.
4. Display the value in dollars and loose change.

**Step 2 Refinement**

2.1. TotalCents is 5 times Nickels plus Pennies.

**Step 3 Refinement**

3.1. Dollars is the integer quotient of TotalCents and 100.

3.2. Change is the integer remainder of TotalCents and 100.

**Ada Source Code**

```
-- CS2970 Case Study 1 ( T. Shimeall)
-- Program to convert pennies and nickels to dollars and cents
-- Program last modified Oct 1994



with Text_IO;
use Text_IO;

procedure PiggyBank is
        Pennies, Nickels : Natural;
        Dollars, Cents   : Natural;
        TotalCents       : Natural;

package My_Nat_IO is new Integer_IO ( Num => Natural);

begin

-- 1 Read in count of Pennies and Nickels
        Put( Item => "Enter number of pennies ");
        My_Nat_IO.Get( Item => pennies);
        Put( Item => "enter number of nickels ");
        My_Nat_IO.Get ( Item => Nickels);

--2 Compute total value in cents
        TotalCents := 5 * Nickels + Pennies;

--3 Find the value in dollars and loose change
```

```
        --3.1 Dollars is the integer quotient of TotalCents and 100
        Dollars := TotalCents / 100;

--3.2 Cents is the integer remainder of TotalCents and 100
        Cents := TotalCents rem 100 ;

--4 Display the value in dollars and loose change
        Put( item => "That works out to ");
        My_Nat_IO.Put( Item => Dollars)
        Put( Item => "dollars and ");
        My_Nat_IO.Put( Item => Cents );
        Put( Item => " cents ");
         New_Line;
end PiggyBank;
```

**Ada Problem Solving**

1. Problem

What is the problem you are required to solve?

2. Analysis

Determine what you are asked to do.

Divide and conquer!

3. Data Requirements and Formulas

What are the required inputs?

What are the required outputs?

What  are the formulas or relationships?

4. Design

Use a stepwise refinement process.

Develop initial algorithm as a series of verbal steps

Refine each verbal step.

Repeat this process until you can put it in Ada.

5. Implementation

Write the algorithm in Ada code.

## Program Reading

Most programming is modifying existing code

- COTS/GOTS
- Reuse libraries
- Maintenance

Need to be able to read a program for understanding

- Error detection/correction (debugging)
- Design a modification
- Review for acceptability / delivery

Reading strategies

- Top-down
- Bottom-up
- Sandwich (mix of top-down & bottom-up)

**Reading Example**

```
-- CS 2970 Bad example 1 ( T. Shimeall)
-- The following was written by a "clever" programmer and is hard
-- to read and prone to mistakes. DON'T use this sort of style if
-- you want a decent grade in CS 1970
-- Program last modified Oct 1994.


with Text_IO;
use Text_IO;
procedure S is
        A, B : Integer;
package I is new Integer_IO( Num => Integer);
begin
        Put( Item => " Enter two integers ");
        I.Get( Item => A);
        I.Get( Item => B);
        New_Line'
        A:= A + B ;
        B:= A - B ;
        A:= A - B;
        Put( Item => " I get ");
        I.Put( Item => B);
        Put( Item => "and");
        I.Put( Item => A);
        New_Line;
end S;
```

**Types revisited**

The real world is filled with various sorts of objects

- Integers ( positive and negative 0 -Ex. account levels
- Natural numbers - Ex. counts
- Fractional numbers - Ex. lengths
- Strings - Ex. proper nouns
- Enumerated values - Ex. military ranks
- etc.

Legal operations on one sort aren't legal on other sorts :

> 5 - 3 = 2
>
> CDR - LCDR = ?

Ada types allow program variables to reflect to sort of objects that they represent

Ada standard environment provides a number of generally useful types :

> Integer, Natural, float, String, Wide_String, Character, Wide_Character

Ada allows programmers to declare their own types (called enumeration)

> Placing limits on existing types
>
> Creating entirely new types

The type declaration is in fact two things :

1. An anonymous type is declared and
2. A subtype with the type name of this anonymous type is declared

**Adding Limits on Existing Types**

For integers, can limit range of values :

- ➢ hours, ranging from 0 to 23
- ➢ days, ranging from 1 to 31
- ➢ etc

**Declaration**

> **subtype** hours **is** integer **range** 0 .. 23;
>
> **subtype** days **is** Integer range 1 .. 31;

For floats, can limit both range and precision:

- ➢ Non-negative floats ( lengths, etc. )
- ➢ Manage significant digits in calculations

**Declaration**

> **subtype** NonNegFloat **is** Float **range** 0.0 .. Float'Last;
>
> **subtype** ApproxFloat **is** Float **digit** 5;

**Attributes of Types**

Attributes ( attribute functions in text ) are bits of information associated with an Ada object.

**Examples:**

| | |
|---|---|
| Integer'First | -- most negative integer |
| Integer'Last | -- most positive integer |
| Float'Last | -- most positive float |
| Float'Large | -- same as Float'Last |
| Float'Digits | -- number of significant digits float can hold |
| Float'Small | -- smallest positive float (nonzero) |
| Character'Pos | -- gives the character code |
| Character'Val | -- gives the character that has certain character code |

We can use these attributes :

- In declarations (see NonNegFloat on previous )
- In calculations or other program statements
- In output

Most attributes are only changed by the way objects are declared ( cannot change after declaration )

**Numerical Input / Output**

Since many numeric types can exist, need to tell Ada which types we want to do input/output with

**First Instantiate ( Tailor ) the Appropriate Packages**

➢ Either precompile the instantiations into our working Ada library

```
-- precompiled instantiations of Integer_IO and
-- Float_IO for the predefined Integer and Float types

with Text_IO;
package My_Int_IO is new Text_IO.Integer_IO( Num => Integer);

with Text_IO;
package My_Flt_IO is new Text_IO.Float_IO( Num => Float);

with Text_IO;
package My_Nat_IO is new Text_IO.Integer_IO( Num => Natural);

```

➢ OR . . .

**Numerical Input/Output**

➢ Add line 5 to programs that do Integer I/O, line 8 to programs that do floating point I/O and line 11 to programs that do Natural I/O.

The Package instantiations would go in the declarations section of your program ( between **procedure** and **begin** ).

```
with Text_IO;
procedure Number_Example is
-- Declaration Area
        Int1, Int2 : Integer;
        Flt1,Flt2   : Float;

package My_Int_IO is new Text_IO.Integer_IO( Num => Integer);

package My_Flt_IO is new Text_IO.Float_IO( Num => Float);

begin  -- Main Program

-- Program Statements

end Number_Example;
```

**Formatted Integer Values**

My_Int_IO.Put( Item =>  n, Width => w);

| Value | Width | Displayed Output |
|---|---|---|
| 234 | 4 | (_)234 |
| 234 | 5 | (_)(_)234 |
| 234 | 6 | (_)(_)(_)234 |
| -234 | 4 | -234 |
| -234 | 6 | (_)(_)-234 |
| 234 | Len | (_)(_)(_)234 if Len is 6 |
| 234 | 1 | 234 |
| 234 | 0 | 234 |

**Formatted Floating-Point Values**

My_Flt_IO.Put ( Item => x, Fore f, Aft => a, Exp => e);

| Value | Fore | Aft | Exp | Displayed Output |
|---|---|---|---|---|
| 3 . 14159 | 2 | 2 | 0 | (_)3.14 |
| 3 . 14159 | 1 | 2 | 0 | 3.14 |
| 3 . 14159 | 3 | 1 | 0 | (_)(_)3.1 |
| 3 . 14159 | 1 | 3 | 0 | 3.14 |
| 3 . 14159 | 2 | 5 | 0 | (_)3.14159 |
| 3 . 14159 | 1 | 3 | 2 | 3.142E+00 |
| 0 . 1234 | 1 | 2 | 0 | 0.12 |
| -0 . 006 | 1 | 2 | 0 | -0.01 |
| -0 . 006 | 1 | 2 | 2 | -6.00E-03 |
| -0 . 006 | 1 | 5 | 0 | -0 . 00600 |
| -0 . 006 | 4 | 3 | 0 | (_)(_)-0.006 |

**Program Documentation**

- Describes our intentions and thought process for the algorithm and the final program.
- Contains information about the program data requirements and algorithm.
- Includes comments in the program

  Problem statement

  Analysis statements

  Algorithm outline

Follow the software development method, then use the documentation developed as starting point in coding your program.

## Developing a Program from its Documentation

Duplicate the problem data requirements (part of the analysis phase) in the program declaration part as comments.

Below each of these comments, enter the Ada syntax for constant and variable declarations.

To develop the program body, begin with the initial algorithm written as a list of comments. Then, move each algorithm refinement under the algorithm step that it refines. After the refinements are in place in the program body, you can begin to write actual Ada statements. Place the Ada code for each step directly under that step. For very simple steps that are refined (equivalent to one Ada statement), edit the refinement, changing it from English to Ada.

## Case Study : Finding Area and Circumference of a Circle

### Problem

Read in the radius of a circle and compute and print its area and circumference.

### Analysis

Clearly, the problem input is the circle radius. Two outputs are requested: the circle area and circumference. These variables should be type Float because the inputs and outputs may contain fractional parts.

**Data Requirements**

**Relevant Formulas**

From our knowledge of geometry, we know the relationship between a circle's radius and its area and circumference; these formulas are listed below. Note that we have written the English description of each variable as an Ada comment to make it easier to produce the declaration part of our solution program.

area of a circle = p x radius$^2$

circumference of a circle = 2 x p x radius

**Problem constant**

```
Pi : constant : Float := 3.14159 ; -- ( Pi  = p )
```

**Problem inputs**

```
Radius : NonNegFloat; --radius of a circle
```

**Problem outputs**

```
Area   : NonNegFloat;  -- area of a circle
Circum : NonNegFloat;  -- circumference of a circle
```

**Initial Algorithm**

1. Read the circle radius
2. Find the area.
3. Find the circumference.
4. Print the area and circumference.

**Step 2 refinement**

2.1. Assign Pi * radius * Radius to Area.

**Step 3 refinement**

3.1. Assign 2 * Pi * Radius to Circum.

**Program Framework**

```
-- CS 2970 Case Study 2
-- Finds and displays the area and circumference of a circle

procedure AreaAndCircum is
        subtype NonNegFloat is Float range 0.0 .. Float'Last;
        Pi : constant : NonNegFloat := 3.14159;

        Radius         : NonNegFloat;  -- input - radius of a circle
        Area           : NonNegFloat;  -- output - area of a circle
        Circumference  : NonNegFloat;  -- output - circumference of a circle

begin -- AreaAndCircum
        null   -- the null makes if compilable - does nothing else
        -- 1 Read the circle radius

        --2 Find the area
        --2.1 assign Pi * radius **2 to Area

        --3 Find circumference
        --3.1 Assign 2.0 * Pi * Radius to Circumference

        --4 Display the Area and Circumference

end AreaAndCircum;
```

Program - Final

```
-- CS 2970 Case Study 2
-- Finds and displays the area and circumference af a circle
-- Program last modified Oct 1994
with Text_IO;
use  Text_IO;
procedure AreaAndCircum is
subtype NonNegFloat is Float Range 0.0 .. Float'Last;
        Pi : Constant NonNegFloat := 3.14159;
        Radius       : NonNegFloat;
        Area        : NonNegFloat;
        Circumference: NonNegFloat;
package My_Fl_Io is new Float_Io( Num => Float);

begin
        put( Item => " Enter radius> ");
        My_Fl_Io.Get ( Item => Radius);
        Area := Pi*Radius**2;
        Circumference := 2.0*Pi*Radius;
        Put( Item => " The area is ");
        My_Fl_Io.Put( Item => Area, Fore => 1, Aft => 2, Exp => 0);
        New_Line;
        Put( Item => " The circumference is ");
        My_Fl_Io.Put( Item => Circumference, Fore =>1, Aft =>2, Exp =>0);
        New_Line;
end AreaAndCircum;
```

**Enumeration Types**

A type is a set of values and a set of operations appropriate for those values.

An enumeration type is a type defined by listing or enumerating its values. These values are called enumeration literals.

Values must be either character literals or identifiers.

The ordering of values in an enumeration type is the order of their appearance in the definition.

Enumeration literals are not case-sensitive.

**Examples**

**type** Day **is** (Monday, Tuesday, Wednesday, Thurshday, Friday,
            Saturday, Sunday);
**type** Colors **is** (Red, Orange, Yellow, Green, Blue, Purple);
**type** NavyRanks **is** (ENS, LTJG, LT, LCDR, CDR, CAPT, R_ADM_L,
                R_ADM_U, V_ADM, ADMIRAL);
**type** ArmyRanks **is** (SCND_LT, FIRST_LT, CPT, MAJ, LTCOL, COL,
                BRIG_GEN, MAJ_GEN, LT_GEN, GENERAL);

**Operations on enumeration Objects**

**Input/Output:**

Done by instantiating (tailoring) the package Text_IO.Enumeration_Io for the enumeration type.

**package** package_name **is new** Text_IO.Enumeration_IO( Enum =>

Our_Enumeration_Type);

**Assignment :**

An enumeration value can be stored into an enumeration variable of that type; one enumeration variable can be copied to another.

**Equality/Inequality Test :**

One enumeration value can be compared to another for equality/inequality ( = , /=)

**Relational operators:**

<, >, <=, >= are all defined for enumeration objects. The ordering relationship corresponds to the order of appearance in the type definition.

**Attribute functions:**

Our_Enumeration_Type'First : the first value in the type

For type Days the Days'First is Monday

Our_Enumeration_Type'Last : the last value in the type

For type NavyRanks the NavyRanks'Last is ADMIRAL

Our_Enumeration_Type'Image : returns as a result a text string in whivh is written the value of the parameter

Our_Enumeration_Type'Value ( inverse of Image (). The parameter text must be of type string and contain text which can be interpreted as a literal of type Our_Enumeration_Type.

Our_Enumeration_Type'Min(X,Y) : returns the lesser of X and Y ( X, Y from type Our_Enumeration_Type).

Our_Enumeration_Type'Max(X,Y) :returns the greater of X and Y

Succ and Pred : successor and predecessor

Pos and Val : position of a given value; value at a given position

## Attribute Functions

```
-- CS 2970 Example 4 ( T Shimeall)
-- Nonsence program to demonstrate attribute functions
-- Program last modified Oct 1994 ( Lt.Col D. Giatros)
with Text_IO;
use  Text_IO;
 procedure Something is
          type Days is ( Monday, Tuesday, Wednesday, Thursday, Friday,
                              Saturday, Sunday);
          Today      : Days;
          AnotherDay: Days;
          Position  : Natural;
 package My_Nat_Io is new Integer_Io( num => Natural);
 package  Days_Io is new Enumeration_Io( Enum => Days);
 begin
          Today := Thursday;
          Position := Days'Pos(Monday);
          My_Nat_Io.Put( Item => Position);
          New_Line;
          AnotherDay := Days'Succ(Today);
          Days_Io.Put( Item => AnotherDay);   New_Line;
          Position := Days'Pos(Today);
          My_Nat_Io.Put( Item => Position);
          New_line;
          AnotherDay := Days'Pred(Today);
          Days_Io.Put( Item => AnotherDay);
```

```
    New_Line;
--  AnotherDay := Days'Succ(Sunday); Raise Constraint error
--  Days_IO.Put( Item => AnotherDay);
--  New_Line;
--  Anotherday := Days'Pred(Monday);
--  Days_Io.Put( Item => AnotherDay);
--  New_Line;
    AnotherDay := Days'Val(6);
    Days_Io.Put( Item => AnotherDay);
    New_Line;   AnotherDay := Days'Val(0);
    Days_Io.Put( Item => AnotherDay);
    New_Line;
--  Position := Days'Val(Days'Succ(Today)); Raise Constrained_Error
--  My_Nat_Io.Put( Item => Position);

end Something;
```

## Case Study : Translating from Arny/Air Force/Marine to Navy/Coast Guard Officer Ranks

### Problem

You are a confused new professor at the Naval Postgraduate School, with students from a mixture of services. You'd like to have an easy way to translate the Army/Air force/marine Officer ranks to their Navy/Coast Guard equivalents.

### Analysis

The two sets of ranks can be represented by two enumeration types NavyRanks and ArmyRanks, and can be read and displayed using two instances of Enumeration_IO, which we will call Navy_IO and Army_IO. We can use attribute functions to do the translation.

**Data Requirements**

**Problem data types :**

Navy/Coast Guard officer ranks, an enumeration type

```
type NavyRanks is (ENS, LTJG, LT, LCDR, CDR, CAPT, R_ADM_L,
                   R_ADM_U, V_ADM, ADMIRAL);
```

Army/Air Force/Marine officer ranks, also an enumeration type:

```
type ArmyRanks is (SCND_LT, FIRST_LT, CPT, MAJ, LTCOL, COL,
                   BRIG_GEN, MAJ_GEN, LT_GEN, GENERAL);
```

**Problem inputs :**

Army/AirForce/Marine rank ( FromRank : ArmyRanks)

**Problem outputs :**

Navy/CoastGuard rank (ToRank: NavyRanks)

**Design**

We were careful to list the Navy and Army ranks in the same order, so given an Army rank, the corresponding Navy rank will be in the same position in its type. This gives us the following algorithm.

**Initial Algorithm**

1. Prompt the user to enter one of the Army ranks , FromRank.
2. Find the corresponding Navy rank, ToRank.
3. Display the Navy rank.

**Algorithm Refinements**

Step 2 refinement

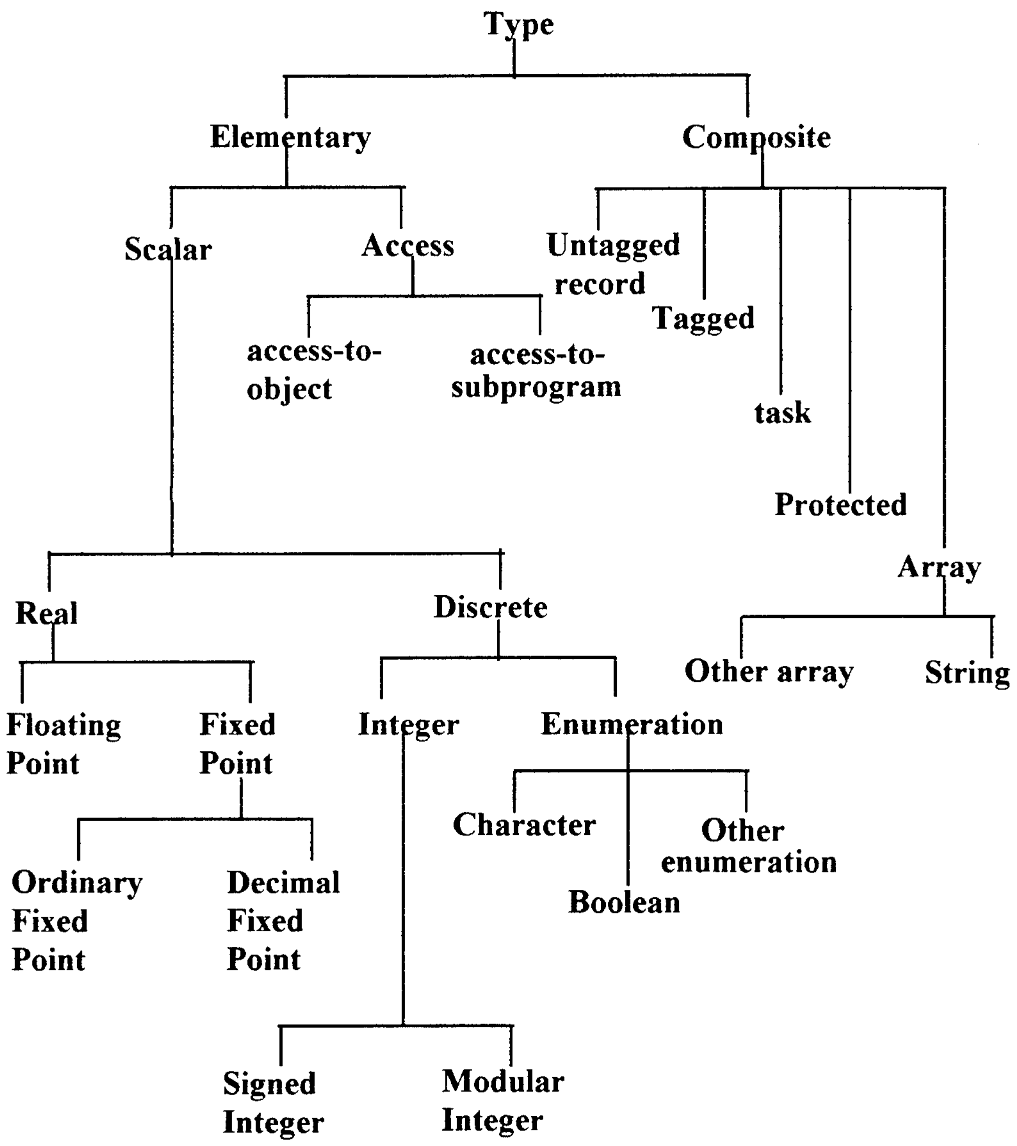
    2.1. Save in Position the position of FromRank in its type.
    2.2. Save in ToRank the corresponding value in the Navy type.

**Program - Final**

```
-- CS 2970 Case Study ( T Shimeall )
-- Convert ranks from Army/Air Force/Marine Officer Ranks to Navy Ranks
-- Program last modified Oct 1994

with Ada.Text_IO;
use  Ada.Text_IO;
procedure ConvertRanks is
        type NavyRanks is (ENS, LTJG, LT, LCDR, CAPT, R_ADM_L,
                                    R_ADM_U, V_ADM, ADMIRAL);
        type ArmyRanks is (SCND_LT, FIRST_LT, CPT, MAJ, LTCOL,
                          COL, BRIG_GEN, MAJ_GEN, LT_GEN, GENERAL);
        FromRank: ArmyRanks;
        ToRank  : NavyRanks;
        Position : Natural;
package Navy_IO is new Enumeration_IO ( Enum => NavyRanks);
package Army_IO is new Enumeration_IO ( Enum => ArmyRanks);
begin
        Put( Item => " Please enter an Army/Air Force/Marine officer rank>");
        Army_IO.Get( Item => FromRank);
        Position := ArmyRanks'Pos(FromRank);
        ToRank := NavyRanks'Val(Position);
        Put( Item => " The Navy/Coast Guard rank is ");
        Navy_IO.Put ( Item => ToRank, Set => Text_Io.Lower_Case);
        New_Line;
end ConvertRanks;
```

Types

- Scalar types - object expressed in a single value.
- Composite types - object is composed of several individual values.
- Discrete - exact values.
- Real - approximate values.



**Subprograms**

**Procedures**

Way to packaging up a series of steps.

We've already used procedures for our sample programs, and you can "nest" procedures by including the full description inside a package or in the declaration section of another subprogram.

Procedures are "called" by naming them and providing any parameters that they need.

```
Put("This is a string")
```

More on defining procedures later.

**Functions**

Way of packaging up a computation to use it multiple times.

Functions always return a value, so they need to be called as part of an expression.

```
Val1:= Val2 + Abs(Val3);
```

More on declaring functions later.

**Tasks**

Left to other courses

**Case Study : Displaying Today's Date in "mm/dd/yy" Form**

**Problem**

Display today's date in the usual American form mm/dd/yy; for example, if today is October 21, 1991, we display 10/21/91. If today is July 8, 1992, we display 7/8/92.

**Analysis**

Somehow we need to be able to ask the computer "what is today's date".

Today's date can be obtained from the computer's internal clock using the appropriate Ada calendar facilities to get a time value and then to extract the month, day, and year. These three values can then be formatted to give the desired display.

**Data Requirements**

Problem data types :

We need only the type Time and the subtypes Year_Number, Month_Number, and Day_Number provided by the standard package Calendar.

**Problem Inputs:**

No inputs need to be entered by the user.

**Problem Outputs:**

Today's date, in the form mm/dd/yy.

**Design**

**Initial Algorithm**

1. Get the current time value from the computer's clock.
2. Extract the current month, day, and year from the time value.
3. Format and display the date.

**Algorithm Refinements**

Step 2 refinement

2.1. Extract the current month from the time value.

2.2. Extract the current day from the time value.

2.3. Extract the current year from the time value.

In step 3, we note that because the year is in the form yyyy (for example 1989), we need to select the last two digits for formatting.

Step 3 refinement

3.1 Find the last two digits of the year.

3.2 Format and display the current month, day, and the last two digits of the year.

**Program - Final**

```
-- Cs 2970 Case Study 4
-- Finds and displays today's date in the form mm/dd/yy
-- The date is gotten from package Calendar
-- Program last modified Oct 1994

with Ada.Text_Io,Ada.Calendar;
use  Ada.Text_Io,Ada.Calendar;

procedure TodayDate is
        Right_Now       : Ada.Calendar.Time;
        This_Year       : Year_Number;
        This_Month      : Month_Number;
        This_Day        : Day_Number;
        Last_Two_Digits : Natural;
        This_Century    : Constant := 1900;
        package My_Nat_Io is new Integer_Io(Num => Natural);

begin

        Right_Now := Ada.Calendar.Clock;
        This_Month := Month( Date => Right_Now);
        This_Day   := Day ( Date => Right_Now);
        This_Year  := Year( Date => Right_Now);
        Last_Two_Digits:= This_Year-This_Century;
        Put ( Item => " Today Date is ");
        My_Nat_IO.Put( Item => This_Month, Width => 1);
        Put( Item => "/");
        My_Nat_IO.Put( Item => This_Day, Width =>1);
        Put( Item => "/");
        My_Nat_IO.Put( Item => Last_Two_Digits, Width =>1);
        New_Line;
end TodayDate;
```

**Program - Alternative**

```
-- CS 2970 Case study 4 - Alternative
-- Finds today's date and displays in the form month dd, yyy
-- An enumeration type is used for months
-- The date is gotten from package Calendar.
-- Program last modified Oct 1994

with Ada.Text_Io,Ada.Calendar;
use  Ada.Text_Io,Ada.Calendar;

procedure TodayDate2 is
        type Months is (January, February, March, April, May, June,
                        July,August,September,October,November,December);
        Right_Now        : Ada.Calendar.Time;
        This_Year        : Year_Number;
        This_Month       : Month_Number;
        This_Day         : Day_Number;
        Last_Two_Digits  : Natural;
        Month_Name       : Months;
        package My_Nat_IO is new Integer_IO(Num => Natural);
        package My_Enum_IO is new Enumeration_IO( Enum => Months);

begin

        Right_Now := Ada.Calendar.Clock;
        This_Month := Month( Date => Right_Now);
        This_Day   := Day ( Date => Right_Now);
        This_Year  := Year( Date => Right_Now);
        Month_Name := Months'Val(This_Month -1);
        Put ( Item => " Today's Date is ");
        My_Enum_IO.Put( Item => Month_Name, Set => Upper_Case);
        Put(",");
        My_Nat_IO.Put( Item => This_Day, Width =>1);
        Put( Item => ",");
        My_Nat_IO.Put( Item => This_Year, Width =>1);
        New_Line;

end TodayDate2;
```

➢ Sample Output

```
Today's Date is October 15, 1994
```

**The If Statement**

```
if Gross > 100.00 then
        Net := Gross - Tax;
else
        Net := Gross;
end if;
```

- ➢ Selects the statement folowing **then** if the Boolean expression is true ( i.e. if Gross is greater than 100.00)
- ➢ Selects the statement following **else** if the Boolean expression is false ( i.e. if Gross is not greater than 100.00).

## Multiple Statements in <u>if</u> statement

As a manager of a clothing boutique, you want to keep records of your bank transactions. You could use the **<u>if</u>** satement below to process a transaction amount (<u>TransAmount</u>) that represents either a payment for goods received ( in which case, <u>Trans Type</u> is 'C') or cash deposit. In either case, an appropriate message is displayed and the account balance (<u>Balance</u>) is updated.

```
if TransType := 'C' then
        Text_IO.Put( Item => 'Check for $");
        My_Flt_IO.Put( Item => TransAmount, Fore => 1, Aft => 2, Exp => 0);
        Text_IO.New_Line;
        Balance := Balance - TransAmount  -- Deduct check amount
else   -- deposit
        Text_IO.Put( Item => 'Deposit of $");
        My_Flt_IO.Put( Item => TransAmount, Fore => 1, Aft => 2, Exp => 0);
        Text_IO.New_Line;
        Balance := Balance + TransAmount  -- Add deposit amount
end if;
```

## The One Alternative if Statement

The following if statement has one alternative, which is executed only when X is not equal to 0.0. It causes Product to be multiplied by X; the new value is then saved in Product, replacing the old value. If X is equal to 0.0 the multiplication is not performed.

```
-- Multiply Product by a nonzero X only
if X /= 0.0 then
        Product := Product * X;
end if;
```

The following if statement orders any two values stored in variables X and Y so that X is not greater than Y. if the two numbers are already in the proper order, the statement sequence is not executed.

```
if X > Y then          --switch X and Y
        Temp := X;  -- Store old X in Temp
        X := Y;     -- Store old Y in X
        Y := Temp;  -- Store old X in Y
end if;
```

The variables X, Y, and Temp must, of course, all be the same type.

**The Multiple - Alternative if Statement**

The if statement below has three alternatives. It causes one of the three variables (NumPos, NumNeg, or NumZero) to be increased by 1 depending on whether X is greater than 0, less than 0, or equal to 0, respectively.

```
-- Increment NumPos, NumNeg, NumZero depending on X
 if   X > 0  then
        NumPos := NumPos + 1;
elsif  X < 0  then
        NumNeg := NumNeg +1;
else  -- X = 0
        NumZero := NumZero + 1;
end if;
```

Here is a four alternative if statement

```
if  GPA <= 1.5  then
        Text_IO.Put( Item => "Exploring civilian opportunities");
elsif  GPA < 2.0  then
        Text_IO.Put( Item => "Academic Probation");
elsif  GPA < 3.0  then
        Text_IO.Put( Item => "Progressing satisfactorily");
else
        Text_IO.Put( Item => "Made the Dean's List- send money");
end if;
```

**An if Statement Implementing a "Decision Table"**

| Salary Range | Base Tax | Percentage of Excess |
|---|---|---|
| 0.00 - 1499.99 | 0.00 | 15% |
| 1500.00 - 2999.99 | 225.00 | 16% |
| 3000.00 - 4999.99 | 465.00 | 18% |
| 5000.00 - 7999.99 | 825.00 | 20% |
| 8000.00 - 14999.99 | 1425.00 | 25% |

```
if Salary < 0.0 then
        Text_IO.Put ( Item => "Error! Negative salary $");
        My_Flt_IO.Put( Item => Salary, Fore => 1, aft => 2, Exp => 0);
        Text_IO.New_Line;
elsif  Salary  < 1500.00 then   -- first range
        Tax := 0.15 * Salary;
elsif  Salary  <  3000.00 then   -- second range
        Tax := ( Salary - 1500.00) * 0.16 + 225.00;
elsif  Salary  <  5000.00 then   -- third range
        Tax := ( Salary - 3000.00) * 0.18 + 465;
elsif  Salary  <  8000.00 then   -- fourth range
        Tax := ( Salary - 5000.00) * 0.20 + 825;
elsif  Salary  <  15000.00 then   -- fifth range
        Tax := ( Salary - 8000.00) * 0.25 + 1425;
else
        Text_IO.Put ( Item => "Error! Too large salary $");
        My_Flt_IO.Put( Item => Salary, Fore => 1, aft => 2, Exp => 0);
        Text_IO.New_Line;
end if;
```

**Be Careful of the Order of Alternatives!**

**wrong!**

```
if Score >= 60 then
        Text_IO.Put ( Item => 'D');
elsif Score >= 70 then
        Text_IO.Put ( Item => 'C');
elsif Score >= 80 then
        Text_IO.Put ( Item => 'B');
elsif Score >= 90 then
        Text_IO.Put ( Item => 'A');
else
        Text_IO.Put ( Item => 'X');
end if;
```

right!

```
if Score >= 90 then
        Text_IO.Put ( Item => 'A');
elsif Score >= 80 then
        Text_IO.Put ( Item => 'B');
elsif Score >= 70 then
        Text_IO.Put ( Item => 'C');
elsif Score >= 60 then
        Text_IO.Put ( Item => 'D');
else
        Text_IO.Put ( Item => 'X');
end if;
```

**If Statement Example**

```
-- CS 2970 Example 5
-- Read in three letters and print the one that comes first, alphabetically
-- Program last modified Oct 1994

with Ada.Text_IO;
use Ada.Text_IO;

procedure FirstLetter is
        Ch1,Ch2,Ch3:Character;  -- input three letters
        AlphaFirst :Character;      -- output alphabetically first letter

begin

        Put( Item => "Enter any three letters, then press Return>");
        Get( Item => Ch1);
        Get( Item => Ch2);
        Get( Item => Ch3);

        -- Save the smaller of Ch1 and Ch2 in AlpaFirst
        if Ch1 < Ch2 then
                AlphaFirst:= Ch1;
        else
                AlphaFirst:= Ch2;
        end if;
        -- Save the smaller of Ch3 and AlphaFirst in AlphaFirst
        if Ch3 < AlphaFirst then
                AlphaFirst:=Ch3;
        end if;

        Put( Item => AlphaFirst);
        Put( Item => " is the first letter alphabetically");
        New_Line;

end FirstLetter;
```

## Functions

- A function is one kind of subprogram, a way of "putting a computation in a box" so it can be done repeatedly.
- Each function is defined to take zero or more parameters, which are its input values, and must also have a return type, indicating the type of value to be returned as the output of each function call.

```
-- function specification
function Average ( X : Float, Y : Float, Z : Float) return Float ;


-- function body
function Average ( X : Float, Y : Float, Z : Float) return Float  is
        Result : Float;
begin  -- Average
        Result := ( X + Y + Z ) / 3.0;
        return Result;
end Average;
```

- X, Y, Z are named formal parameters
- Strictly speaking, the function spec is not necessary when declaring a function in a main program, but we show both, to make it easier to understand package specs and package bodies, which contain function specs and function bodies, respectively.

## Declaring Functions

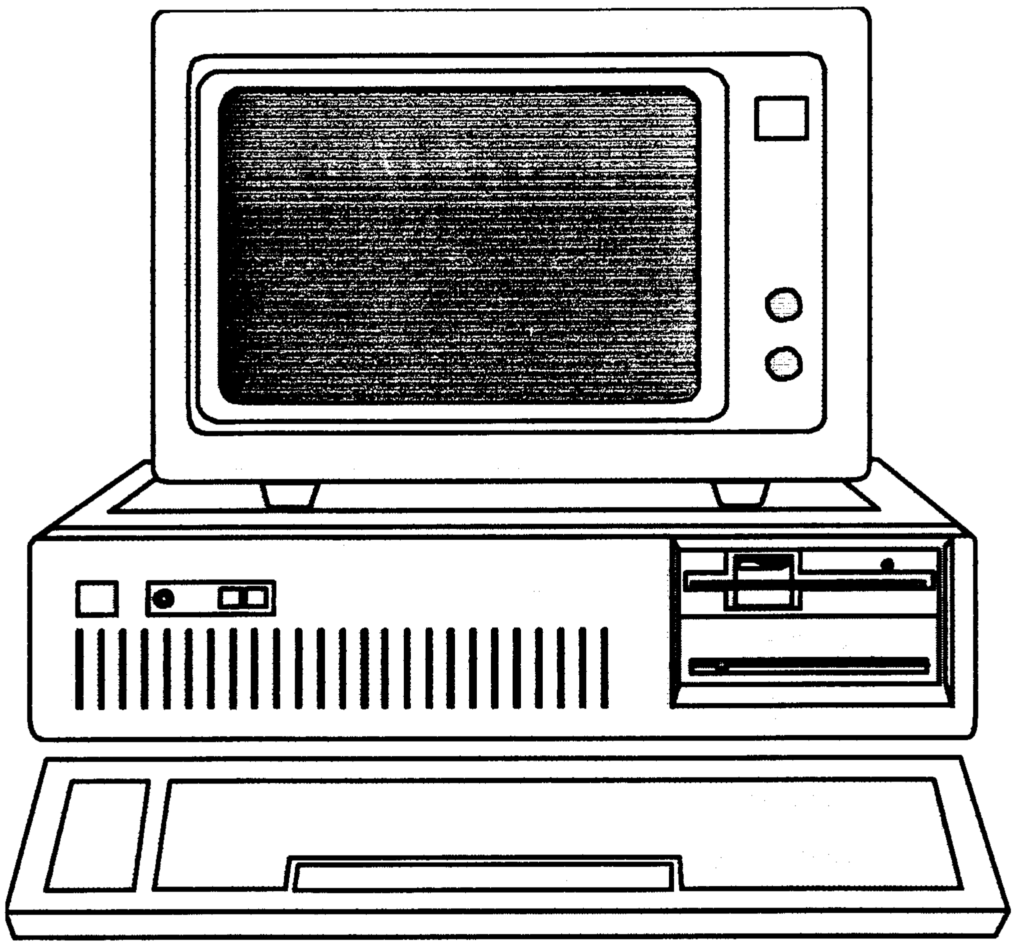
- You can declare a function anywhere you can declare variables, for example, in a main program. The function is then usable only within the program in which it is declared.
- A more useful place to declare a function is in a package, which is then compiled and placed in your library for further use by any other unit ( package or program ) that "**with**"s the package.
- The declaration of a function ( or procedure ) can be placed in arbitrary order in relation with the programs declarations.
- By client we mean a program unit that "**with**"s a package.

```
-- function specification
function Maximum ( Value1, Value2 : Integer) return Integer;


-- function body
function Maximum ( Value1, Value2 : Integer) return Integer is
        Result : Integer;
begin  -- Maximum
        if Value1  >  Value2 then
                Result := Value1;
        else
                Result := Value2
        end if;
        return Result;
end Maximum;
```

**Function Example**

```
-- CS 2970 Example 6 ( T. Shimeall)
-- A simple example of a function declaration and use showing three forms
-- of call syntax
-- This program last modified Oct 1994 by Lt. Col. D. Giatros

with Ada.Text_I0,Ada.Numerics.Elementary_Functions;
use Ada.Text_IO,Ada.Numerics.Elementary_Functions;

procedure Func_Example is
        Side1, Side2 : Float;
        Side3        : Float;

        function Hype_Length ( Leg1,Leg2: Float) return Float is
        begin
                return Sqrt((Leg1*Leg1)+(Leg2*Leg2));
        end Hype_Length;

package My_Fl_IO is new Float_IO( Num => Float);

begin

        Put( Item => "Enter lengths of two sides of a triangle");
        My_Fl_IO.Get( Item=> Side1);
        My_Fl_IO.Get( Item=> Side2);
        New_Line;
        Side3:= Hype_Length(Side1,Side2);
        Put(" The third side has length ");
        My_Fl_Io.Put( Item => Side3, Fore =>0, Aft => 2, Exp => 0);
        New_Line;
    end Func_Example;
```

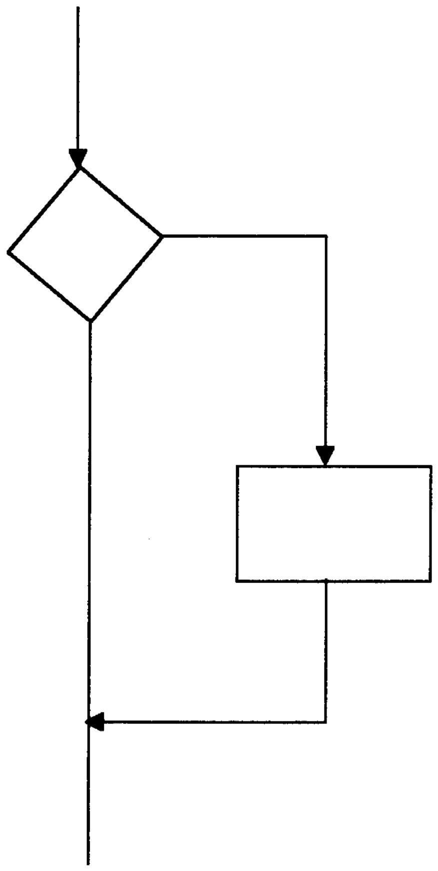
Sample Lab Computer



## Control Statements

➢ Selection : the **if** statement if then

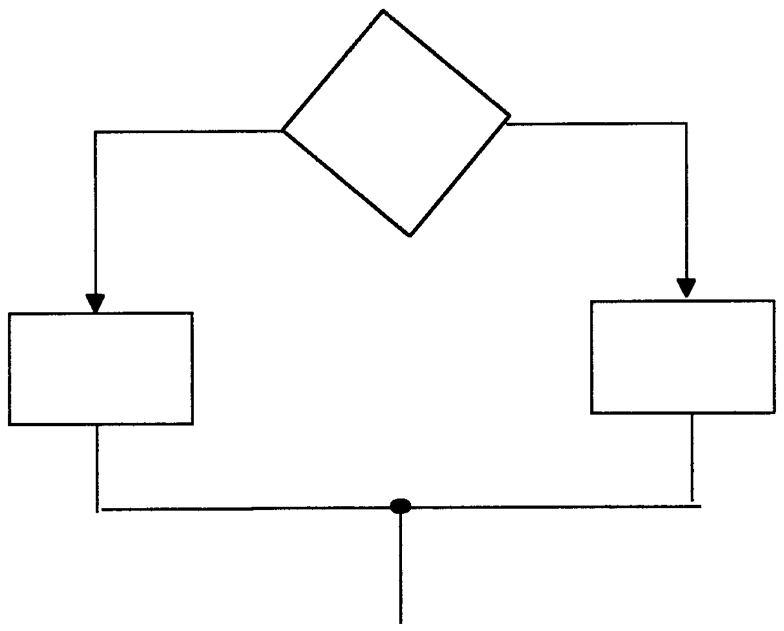
```
if Boolean_expression then
        sequence_of_statements;
end if;
```



**Control Statements**

➢ Selection : the **if** statement If then else

```
if Boolean_expression then
        sequence_of_statements;
else
        sequence_of_statements;
end if;
```



**The If Statement**

```
if Gross > 100.00 then
        Net := Gross - Tax;
else
        Net := Gross;
end if;
```

- ➢ Selects one of the two assignment statements listed. It selects the statement following **then** if the Boolean expression is true ( i.e. if Gross is greater than 100.00)
- ➢ Selects the statement following **else** if the Boolean expression is false ( i.e. if Gross is not greater than 100.00).

**The if statement**

As a manager of a clothing boutique, you want to keep records of your bank transactions. You could use the **if** statement below to process a transaction amount (TransAmount) that represents either a payment for goods received ( in which case, Trans Type is 'C') or cash deposit. In either case, an appropriate message is displayed and the account balance (Balance) is updated.

```
if TransType := 'C' then
        Text_IO.Put( Item => 'Check for $");
        My_Flt_IO.Put( Item => TransAmount, Fore => 1, Aft => 2, Exp => 0);
        Text_IO.New_Line;
        Balance := Balance - TransAmount  -- Deduct check amount
else  -- deposit
        Text_IO.Put( Item => 'Deposit of $");
        My_Flt_IO.Put( Item => TransAmount, Fore => 1, Aft => 2, Exp => 0);
        Text_IO.New_Line;
        Balance := Balance + TransAmount  -- Add deposit amount
end if;
```

**The if Statement**

The following **if** statement has one alternative, which is executed only when X is not equal to 0.0. It causes Product to be multiplied by X; the new value is then saved in Product, replacing the old value. If X is equal to 0.0 the multiplication is not performed.

```
-- Multiply Product by a nonzero X only
if X /= 0.0 then
        Product := Product * X;
end if;
```

The following if statement orders any two values stored in variables X and Y so that X is not greater than Y. if the two numbers are already in the proper order, the statement sequence is not executed.

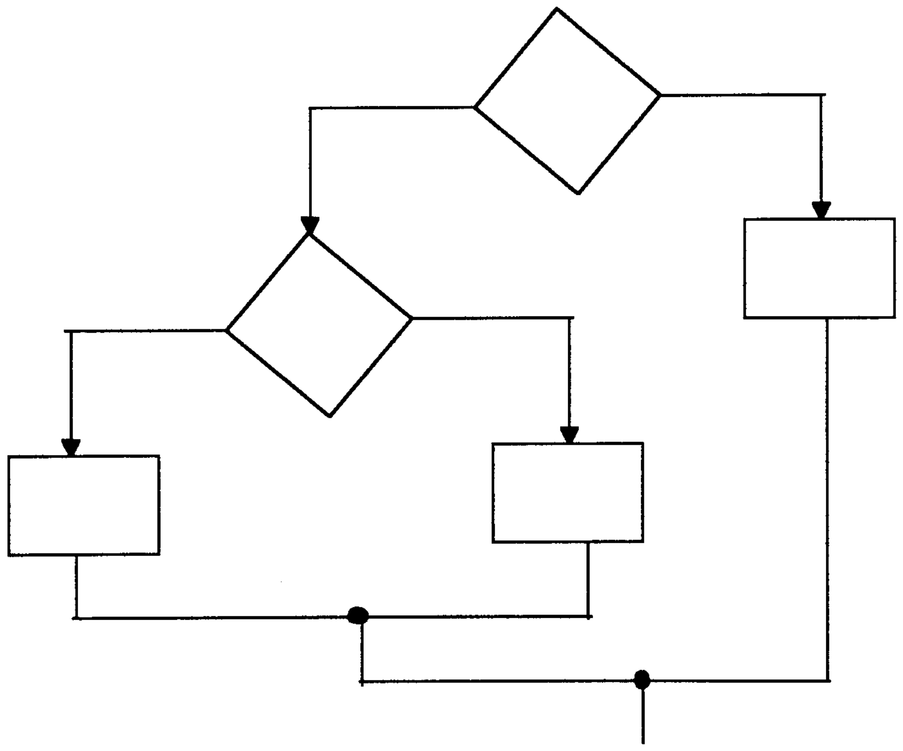
```
if X > Y then          --switch X and Y
        Temp := X;  -- Store old X in Temp
        X := Y;     -- Store old Y in X
        Y := Temp;  -- Store old X in Y
end if;
```

The variables X, Y, and Temp must, of course, all be the same type. Although the values of X and Y are being switched, an additional variable Temp is needed for storage of a copy of one of these values.

**Control Statements**

➢ Selection : the **if** statement **if then elsif else**

```
if Boolean_expression then
        sequence_of_statements;
elsif Boolean_expression then
        sequence_of_statements;
elsif Boolean_expression then
        sequence_of_statements;
. . .
else
        sequence_of_statements;
end if;
```



**The Multiple - Alternative if Statement**

The if statement below has three alternatives. It causes one of the three variables (NumPos, NumNeg, or NumZero) to be increased by 1 depending on whether X is greater than 0, less than 0, or equal to 0, respectively.

```
-- Increment NumPos, NumNeg, NumZero depending on X
if   X > 0  then
         NumPos := NumPos + 1;
elsif  X < 0  then
         NumNeg := NumNeg +1;
else  -- X = 0
         NumZero := NumZero + 1;
end if;
```

Here is a four alternative if statement

```
if GPA <= 1.5  then
         Text_IO.Put( Item => "Exploring civilian opportunities");
elsif  GPA < 2.0  then
         Text_IO.Put( Item => "Academic Probation");
elsif  GPA < 3.0  then
         Text_IO.Put( Item => "Progressing satisfactorily");
else
         Text_IO.Put( Item => "Made the Dean's List- send money");
end if;
```

**Implementing a "Decision Table"**

| Salary Range | Base Tax | Percentage of Excess |
|---|---|---|
| 0.00 - 1499.99 | 0.00 | 15% |
| 1500.00 - 2999.99 | 225.00 | 16% |
| 3000.00 - 4999.99 | 465.00 | 18% |
| 5000.00 - 7999.99 | 825.00 | 20% |
| 8000.00 - 14999.99 | 1425.00 | 25% |

```
if Salary < 0.0 then
        Text_IO.Put ( Item => "Error! Negative salary $");
        My_Flt_IO.Put( Item => Salary, Fore => 1, aft => 2, Exp => 0);
        Text_IO.New_Line;
elsif  Salary  < 1500.00 then   -- first range
        Tax := 0.15 * Salary;
elsif  Salary  <  3000.00 then   -- second range
        Tax := ( Salary - 1500.00) * 0.16 + 225.00;
elsif  Salary  <  5000.00 then   -- third range
        Tax := ( Salary - 3000.00) * 0.18 + 465;
elsif  Salary  <  8000.00 then   -- fourth range
        Tax := ( Salary - 5000.00) * 0.20 + 825;
elsif  Salary  <  15000.00 then   -- fifth range
        Tax := ( Salary - 8000.00) * 0.25 + 1425;
else
        Text_IO.Put ( Item => "Error! Too large salary $");
        My_Flt_IO.Put( Item => Salary, Fore => 1, aft => 2, Exp => 0);
        Text_IO.New_Line;
end if;
```

## Control Statements

➢ Selection : the **case** statement

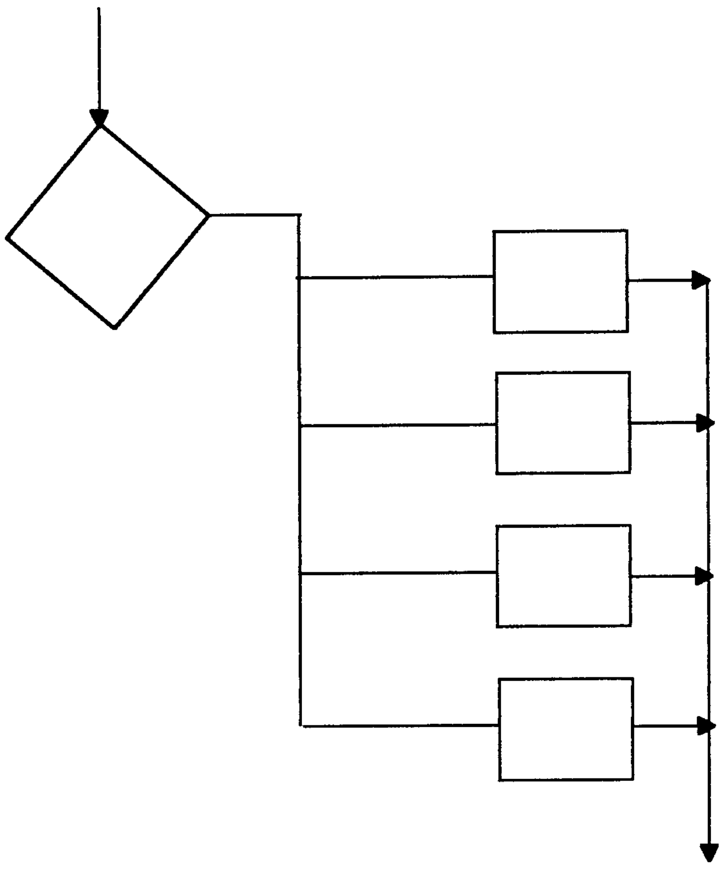
```
case selector is
        when list_of_alternatives =>
                sequence_of_statements;
        when list_of_alternatives =>
                sequence_of_statements;
        . . .
        when others  =>
                sequence_of_statements;

end case;
```

➢ If the selector is a name ( type conversion or a function call ) then each non-others discrete choice shall cover only values in the subtype, and each value of that subtype shall be covered by some discrete choice ( either explicitly or by **others** ).

➢ If the selector is root integer, universal integer or a discriminant of a formal scalar type, then we shall have an others discrete choice.

➢ Otherwise each value of the base range of the type of the expression shall be covered ( either explicitly or by others).

**Case Statement Continuous**



**Control Statements**

➢ Selection : the case statement - alternative list

All possible values must be referenced

```
when 5 =>
when 4 | 8 | 23 =>
when 100 .. 125 =>
when 50 | 60 | 70 .. 75 | 80 .. 85 =>
when others =>
```

Others alternative must come last.

Others alternative can be used as we referred in previous pages.

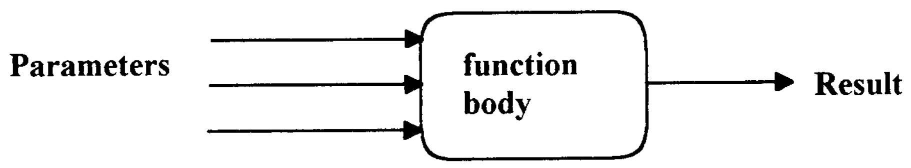
Program - Final

```
-- CS 2970 Example
-- A simple example of a case control statements by using as a selector
-- a function in the main procedure .
-- Program created in Oct 1994

with Ada.Text_IO;
use Ada.Text_IO;

procedure Your_Grade is
        Grade :Integer;
package Integer_IO is new Integer_IO( Num => Integer);
use Integer_IO;

        function Find_Gr( G : Integer) return Character is
                Temp : Character;
        begin
                case G  is
                        when  95..100 => Temp:= 'A';
                        when  85..94 =>  Temp:='B';
                        when  75..84 =>  Temp:='C';
                        when  65..74 =>  Temp:='D';
                        when others =>   Temp:='F';
                end case;
                return Temp;
        end Find_Gr;
```

```
begin    -- Your_Grade

        Put( Item => " Give the grade as integer number");
        New_Line;    Integer_Io.Get( Item => Grade );

        case Find_Gr(Grade) is
                when 'A' => Put(" Your Grade is A ");
                when 'B' => Put(" Your Grade is B ");
                when 'C' => Put(" Your Grade is C ");
                when 'D' => Put(" Your Grade is D");
                when 'F' => Put(" You failed ");
                when others => Put( " There is an error in the program ");
        end case;
        New_Line;

end  Your_Grade;
```

**Functions**

- A function is one kind of subprogram, a way of "putting a computation in a box" so it can be done repeatedly.
- Each function is defined to take zero or more parameters, which are its input values, and must also have a return type, indicating the type of value to be returned as the output of each function call.



## Functions

- Strictly speaking, the function spec is not necessary when declaring a function in a main program.
- Show both, to make it easier to understand package specs and package bodies, which contain function specs and function bodies, respectively.

```
-- function specification
function Average ( X : Float, Y : Float, Z : Float) return Float ;

-- function body
function Average ( X : Float, Y : Float, Z : Float) return Float  is
        Result : Float;
begin  -- Average
        Result := ( X + Y + Z ) / 3.0;
        return Result;
end Average;
```

**Calling A Function**

- Our function specification

    function Maximum ( Value1, Value2 : Integer ) return Integer

    Value1, Value2 are named formal parameters.

- Typical function calls.

    Larger := Maximum( Value1 => FirstValue, Value2 => SecondValue);

    FirstValue, SecondValue are named actual parameters.

Actual parameters bay be expressions and their types must agree with those of the corresponding formal parameters.

## Declaring Functions

- You can declare a function anywhere you can declare variables, for example, in a main program. The function is then usable only within the program in which it is declared.
- A more useful place to declare a function is in a package, which is then compiled and placed in your library for further use by any other unit ( package or program ) that "**with**"s the package.
- The declaration of a function ( or procedure ) can be placed in arbitrary order in relation with the programs declarations.
- By client we mean a program unit that "**with**"s a package.

**Example**

```
-- CS 2970
-- A simple example of a procedure that calls a function to find the larger integer
-- Program last modified Oct 1994

with Text_IO;
use Text_IO;

procedure MaxTwo is

        FirstValue : Integer;
        SecondValue : Integer;
        Larger       : Integer;

package Integer_IO is new Integer_IO( Num => Integer);
use Integer_IO;

-- function specification
function Maximum ( Value1, Value2 : Integer) return Integer;

-- function body

function Maximum ( Value1, Value2 : Integer) return Integer is
        Result : Integer;

begin  -- Maximum
        if Value1  >  Value2 then
                Result := Value1;
```

```
          else
                    Result := Value2
          end if;
          return Result;
end Maximum;

begin -- Maxtwo

          Put( Item => "Please enter first integer value >");
          Integer_IO.Get ( Item => FirstValue);
          Put( Item => "Please enter second integer value >");
          Integer_IO.Get ( Item => SecondValue);

          Larger := Maximum ( Value1 => FirstValue, Value2 => SecondValue);

          Put( Item => " The larger number is ");
          Put( Item => Larger, Width => 1);
          New_Line;

end MaxTwo;
```

## Packages

- A package is Ada's way of letting us collect together, in one place, a number of reusable resources ( procedures, functions, variables, types ) for further use.

- Text_IO and Calendar are two of Ada's predefined packages ( that is required by the standard ). We will use others and also write a few of our own.

- A package has two parts, which are best located in separate files : the package specification and the package body.

- The package specification shall not have a body unless it requires one.

## Categories of Packages

- Packages of types and constants. These types of packages are not allowed to have body.
- Packages with subprograms that logically belong together.
- Packages with memory
- Packages witch construct abstract data types.
- Child packages.

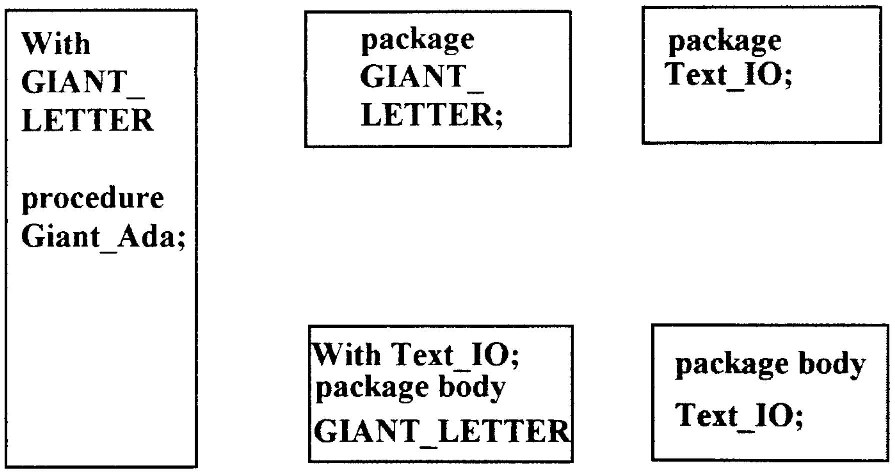
## Separating Package Specs and Bodies

- ➢ An advantage of separating package specification and body files is that one can create several package bodies for a given specification, compiling whichever one is most useful.

- ➢ Switching package bodies ( compiling a different one ) does not necessitate recompiling client programs.

**Package**

GIANT LETTERS



**Packages**

- The package specification serves as a "table of contents" or "contract" with the client program, describing those things in the package that are made available to client programs.

- Entities declared in the package specification are visible to another program unit.

```
package minmax is
-- specifications of functions provided by MinMax package
        function Minimum ( Value1, Value2 : Integer) return Integer;
        function Maximum ( Value1, Value2 : Integer) return Integer;

end minmax;
```

**Packages**

- The package body contains the code bodies for all the procedures and functions promised by the specification.
- Entities declared in the package body are visible only in this package( are not visible to the another program unit)

```
package body minmax is
-- bodies of functions provided by minmax package


    function Minimum( Value1, Value2 : Integer) return Integer is
        Result : Integer;

    begin
        if Value1 < Value2 then
            Result := Value1;
        else
            Result := Value2;
        end if;
        return Result;
    end minimum;
```

```
        function Maximum ( Value1, Value2 : Integer) return Integer is
                Result : Integer;
        begin
                if Value1 > Value2 then
                        Result := Value1;
                else
                        Result := Value2;
                end if;
                return Result;
        end Maximum;
end minmax;
```

**Packages Client Program**

```
with Ada.Text_IO, minmax;
use Ada.Text_Io;

procedure MinMaxThree is

-- finds the largest and smallest of three values
--using the Minimum and Maximum functions from package minmax

        Num1, Num2, Num3,   Largest, Smallest : Integer;

        package Integer_IO is new Integer_IO ( Num => Integer);

begin

        Put ( Item => " Please enter first integer value >");
        Integer_IO.Get( Item => Num1);
        Put ( Item => " Please enter second integer value > ");
        Integer_IO.Get( Item => Num2);
        Put ( Item => " Please enter third integer value >");
        Integer_IO.Get( Item => Num3);
```

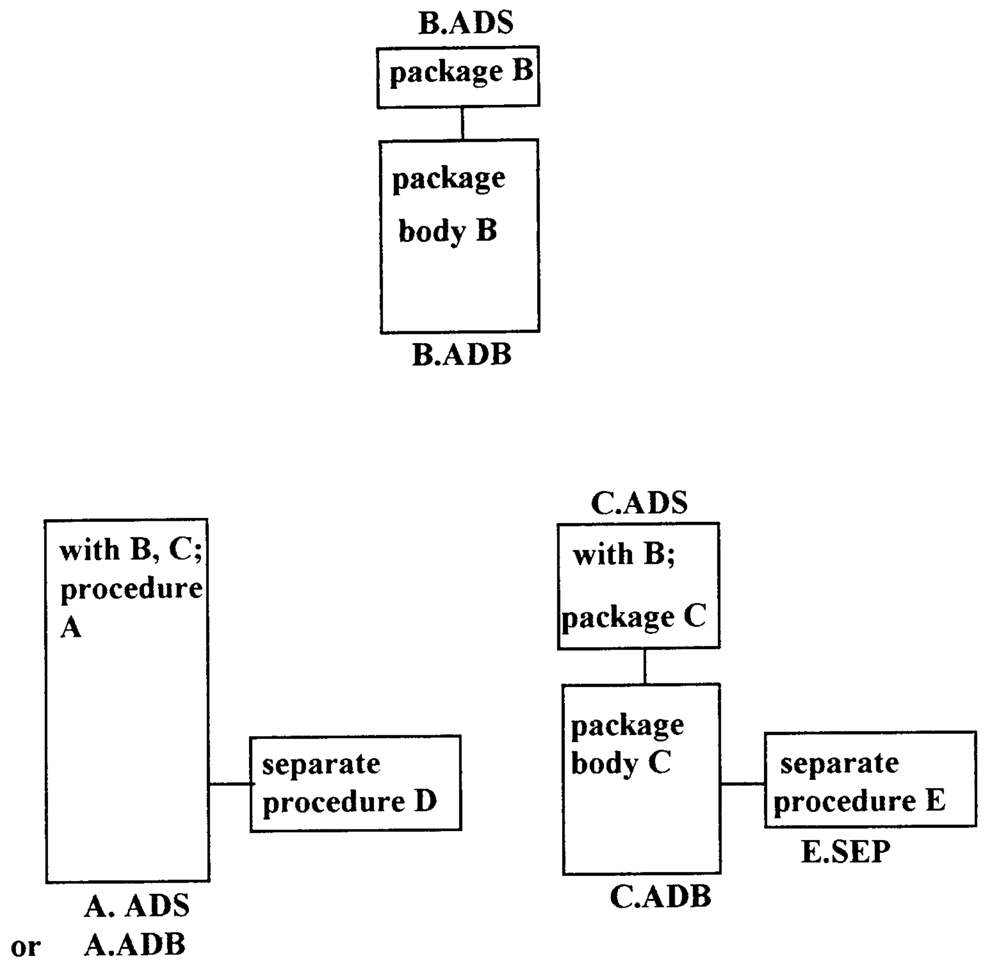
```
    Largest := MinMax.Maximum( Value1 => Num1, Value2 => Num2);
    Largest := MinMax.Maximum( Value1 => Largest, Value2 => Num3);
    Smallest := MinMax.Minimum( Value1 => Num1, Value2 => Num2);
    Smallest := MinMax.Minimum( Value1 => Smallest, Value2 => Num3);

    Put ( Item => " The smallest number is ");
    Integer_IO.Put( Item => Smallest, Width => 1);
    Put ( Item => " and the largest number is ");
    Integer_IO.Put( Item => Largest, Width => 1);
    New_Line;
end MinMaxThree;
```

## Compilation Order

- When working with packages, the order of compilation of the pieces is very important.
- The package specification must be compiled before either the package body or the client, because the compiler checks both package body and client for consistency with the specification.
- The package body does not have to be compiled before the client: all three ( specification, body, client ) must be compiled before linking is possible.
- Recompiling a package body makes it necessary to re-link the client in order to include the modified code in the package body. ( You can continue to use the old executable file, but of course it does not use the latest version of the package).
- Recompiling a package specification makes it necessary to re-compile both the package body and all clients that "with" the package.
- We can avoid a recompilation if we use child packages.( We see later in the Object Oriented programming features).

**Packages**

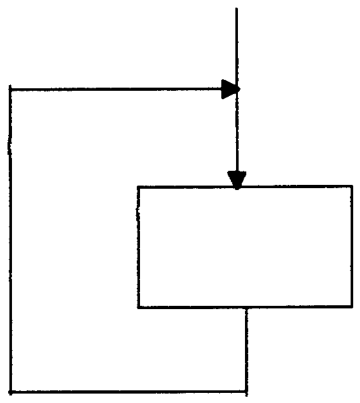
Compilation Order



## Control Statements

➢ **Iteration :** the simple loop statement

```
loop
        sequence_of_statements;
end loop;
```



➢ loop statement includes a sequence of statements that is to be executed repeatedly zero or more times.

**Control Statements**

➢ **Iteration** : the loop statement with for

> **for** loop_parameter **in** start_value **..** end _value **loop**
>
> sequence_of_statements;
>
> **end loop;**

➢ loop_parameter is declared automatically type depends on start_value and end_value.

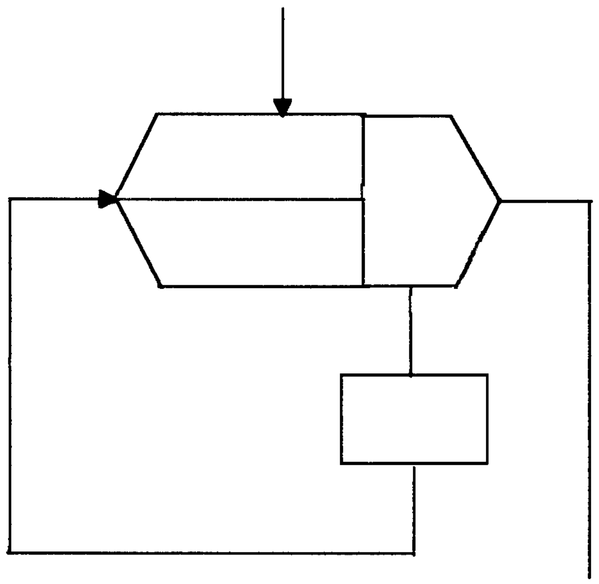
➢ loop_parameter can not be assigned a new value in the loop.

➢ loop_parameter can not be used outsideof the loop

➢ start_value, end_value should be integer type or enumeration type

Examples

> **for** I **in** 1 .. 10 **loop**
>
> **for** I **in** -1 .. 10 **loop**



**Example**

The statement

```
for Count in 1 .. 5 loop
    Text_IO.New_Line;
end loop;
```

has the same effect as the five statements

```
Text_IO.New_Line;
Text_IO.New_Line;
Text_IO.New_Line;
Text_IO.New_Line;
Text_IO.New_Line;
```

The following for loop displays a sequence of HowMany aterisks. If HowMany has a value of 5, 5 asterisks in a row will be displayed; if HowMany has a value of 27, 27 aterisks will be displayed, and so on.

```
for Count in 1 .. HowMany loop
    Text_IO.Put( Item => '*');
end loop;
```

**Example**

```
for Count in  0 .. 255 loop
        if Count mob 8 = 0 then
                Text_IO.New_Line;
        end if;
        Text_IO.Put ( Character'Val( Count));
end loop;


Get ( Value );
for Letter in reverse 'a' .. 'z' loop
        Text_Io.Put(Letter);
        for Space in 2 .. Value loop
                Text_IO.Put( '*');
        end loop;
        Text_IO.New_Line;
end loop;
```

## Case Study : Sum of Integers

**Problem**

Write a program that finds the sum of all integers from 1 to N.

**Analysis**

In order to solve this problem, it will be necessary to find some way to form the sum of the first N positive integers.

**Data Requirements**

**Problem inputs**

The last integer in the sum ( N : Positive )

**Problem outputs**

The sum of integers from 1 to N ( Sum : Natural )

**Design**

**Initial algorithm**

1. Prompt the user for the last integer (N).
2. Find the sum ( Sum) of all the integers from 1 to N inclusive
3. Display the sum

**Algorithm Refinements**

Step 2 refinement

```
    2.0 Set Sum to zero
    2.1 Add 1 to Sum
    2.2 Add 2 to Sum
         . . .
    2.N Add N to Sum
```

For a large value of N it would be rather time consuming to write the list of steps. We would also have to know the value of N before writing this list; consequently, the program would not be general, as it would work for only one value of N.

**Design ( Cont'd)**

Because steps 2.1 through 2.N are all quite similar, we can represent each of them with the general step.

2.1 Add i to Sum

This general step must executed for all values of i from 1 to N, inclusive. This suggest the use of a counting loop with i as the loop variable.

**Program Variables :**

loop control variable - represents each integer from 1 to N ( i : Positive )

The variable i will take on the successive values 1, 2, 3, 4, . . ., N. Each time the loop is repeated, the current value of i must be added to Sum. We now have a new refinement of step 2.

**Step 2 refinement**

```
2.1     for each integer i from 1 to N
            Add i to Sum
        end loop;
```

**Program - Final**

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure SumIntegers is

-- finds and displays the sum of all positive integers from 1 to N
        N    : Positive;
        Sum : Natural;

        package Integer_IO is new Integer_IO ( Num => Natural);

begin  -- SumIntegers

-- Read the last integer N
        Put ( Item => "Enter the last integer in the sum >";
        Integer_IO.Get ( Item => N);

-- Find the sum ( Sum ) of all integers from 1 to N
        Sum := 0;        -- intitialize Sum to 0

        for I in 1..N loop
                Sum := Sum + I;        -- Add the next integer to Sum
        end loop;
```

```
-- Displays the sum
         Put ( Item => "The sum of the integers from 1 to ");
         Integer_IO.Put( Item => N, Width => 1);


         Put ( Item => " is ");
         Integer_IO.Put( Item => Sum, Width => 1);
         New_Line;
end SumIntegers;
```

## More Generalizing : Minimum, Maximum, and Average of a list of Numbers

**Problem :**

Write a program that finds and displays the minimu, maximum, and taverage of a list of integers.

**Analysis**

This is quite similar to the previous problem. We can the variables CurrentValue and Sum as above. As each value is read, it must be added into the sum, but also compared against the current minimum, Smallest, and the current maximum Largest. The comparison can be handled by the Minimum and Maximum functions already provided in the minmax package.

## More Generalizing : Minimum, Maximum, and Average of a list of Numbers

### Data Requirements

**Problem inputs**

Number of items to be averaged

NamValues : Positive

Temporary storage for each data value

CurrentValue : Integer

**Problem outputs**

Minimum of the NumValues data values

Smallest : Integer

Largest of the NumValues data values

Largest : Integer

Average of the NumValues data values

Average : Integer

**Initial Algorithm**

1. Prompt the user for the number ( NumValues ) of values to be summed.

2. Prompt the user for each data value : add it to the sum, check to see if it is a new minimum, maximum, and check to see if it is a new maximum.

3. Compute the average of the values.

4. Display the minimum maximum, and average.

**Initial Algorithm**

Step 2 refinement

2.1. Initialize Sum to 0, Smallest to Integer'Last, and Largest to Integer'First.

```
2.2.
    for each data value loop
        Read the data value in to the CurrentValue and add
        CurrentValue to Sum;
        Determine whether the data value is a new minimum or
        maximum.
    end loop
```

Step 2.2 refinement

```
    for each data value loop
        2.2.1 Read the data value in to the CurrentValue and add
              CurrentValue to Sum;
        2.2.2 Replace Smallest with the smaller of itself and
              CurrentValue.
        2.2.3 Replace Largest with the larger of itself and
              CurrentValue.
    end loop
```

**Program - Final**

```
with Ada.Text_IO,Minmax;
use Ada.Text_IO;

procedure MinMaxAvg is

-- Finds and displays the minimum, maximum and average
-- of a list of data items from an external data file

        NumValues        : Positive;  -- the numeber of items to be averaged
        Sum,                          -- the sum being accumulated
        CurrentValue,                 -- the next data item to be added
        Smallest,                     -- minimum of the data values
        Largest,                      -- maximum of the data values
        Average          : Integer;   -- average of the data values
        TestScores       : File_Type; -- program variable naming th input file

package My_In is new Integer_IO( Num => Integer);

begin

-- Open the file and associate it with the file variable name
        Open ( File => TestScores, Mode => In_File, Name => "scores.dat");
```

```
-- Read from the file the number of items to be averaged
        My_In.Get( File => TestScores, Item => NumValues);
        Put(" The number of scores to be averaged is ");
        My_In.Put( Item => NumValues, Width =>1);
        New_Line;

-- Initialize program variables
        Smallest := Integer'Last;
        Largest  := Integer'First;
        Sum      := 0;


-- Read each data item log to the screen, add it to Sum
-- and check if it is a new minimu or maximum

        for Count in 1 .. NumValues loop
                My_In.Get( File => TestScores, Item => CurrentValue);
                Put( Item => " Score number ");
                My_In.Put( Item => Count, Width => 1);
                Put( Item => " is ");
                My_In.Put( Item => CurrentValue, Width => 1);
                New_Line;
                Sum := Sum + CurrentValue;
                Smallest := minmax.Minimum( Value1 => Smallest, Value2 =>
                                                        CurrentValue);
                Largest  := minmax.Maximum( Value1 => Largest, Value2 =>
                                                        CurrentValue);
        end loop;

-- compute the average since Sum and NumValues are integers
-- the average will be rounded to the nearest integer
        Average := Sum / NumValues;



-- Display the results


        Put( Item => " The Smallest is ");
        My_In.Put( Item => Smallest, Width => 1);
        New_Line;
        Put( Item => " The Largest is ");
        My_In.Put( Item => Largest, Width => 1);
        New_Line;
        Put( Item => " The average is ");
        My_In.Put( Item => Average, Width =>1);
        New_Line;

end MinMaxAvg;
```

## Subtypes of Scalar Types

- ➢ Subtypes let us take advantage of Ada's range checking to detect bad user input.

- ➢ Subtypes also help us more closely model the "real world".

- ➢ General form of a subtype declaration:

```
subtype  name  is  base_type  range  min..max;
        sequence_of_statements;
```

- ➢ Example of general form :

```
subtype  Uppercase  is  character  range  'A' .. 'Z';
```

**Examples of Scalar Subtypes**

➢ Predefined Subtypes in the Language

```
subtype Natural is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```

➢ Predefined Subtypes in the Calendar Package

```
subtype Year_Number is Integer range 1901..2099;
subtype Month_Number is Integer range 1..12;
subtype Day_Number is Integer range 1..31;
```

➢ Subtype of an enumerated type

```
type Days is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    subtype Weekday is Days range Mon..Fri;
    subtype Weekend is Days range Sat..Sun;
    subtype Mon_Fri is Weekday;
```

➢ Other subtypes

```
subtype Inat is Integer;
subtype Flt is Float;
```

➢ Note that the ordering relation is preserved!!

## Compatibility of Types and Subtypes

➢ Two values are compatible if :

1. They have the same type name ( Integer, Integer).
2. One value's type is a subtype of the other's value type ( Natural, Integer).
3. The values are subtypes of the same base type ( Positive, SmallInt).

   [in a limited sense]

➢ That means that even though My_Int_IO.Put ( or Integer_IO.Put) expects an Integer it will accept any subtype of Integer.( e.g. Natural, Positive)

## Type Membership : The Operator in

➢ The **in** Operator can be used to determine whether a given value is a member of a type's set of values.

**Examples**

```
type  Days  is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype  Weekday  is  Days  range  Mon..Fri;
subtype  Weekend  is  Days  range  Sat..Sun;
subtype  Mon_Fri  is  Weekday;

if  Tomorrow  in  WeekDays  then
        Text_IO.Put ( Item => "Another day, another dollar  ");
        Text_IO.New_Line;
else
        Text_IO.Put ( Item =>  "we've worked hard, It's play hard!");
        Text_IO.New_Line;
end if;


package Day_IO is new Enumeration_IO ( Enum => Days);
for  WhichDay  in  Weekdays  loop
        Day_IO.Put ( Item  => WhichDay);
        Text_IO.New_Line;
end loop;
```

## Overloading

- Overloading permits us to bind the same identifier to two or more functions, procedures or operators.

- The advantage is that we can use the same identifier to perform similar operations on different types.

- For example we know we can add two integers using the '+' operator and we can also add two floats using the '+' operator. This is an example of "overloading".

```
a, b, c  : Integer;
x, y, z  : Float;

c := a + b;
z := x + y;
```

- Carrying the idea a step further : if we tell the computer how to do it we can also add two vectors using the same '+' symbol. One we do that we can perform vector addition using the '+' operator. Therefore we have used the same symbol '+' to express the same abstraction in this case addition.

```
a, b, c : Vector;
c := a + b;
```

**Overloading**

- You should note : however, that since YOU are the one who tells the computer HOW to perform addition it does not necessarily mean you will perform addition ( you may actually multiply, divide, or subtract instead).

- Overloading can be abused if it is used too much or if it is used to name functions and procedures that do not have similar behavior ( i.e. we expect a sort procedure to sort and not shuffle data instead).

- Explicit overloadings of " = " are permitted for any combination of parameter and result types.

- Explicit overloadings of " /= " are permitted, so long as the result type is not Boolean.

## Introduction to Exception Handling

- Exception handling allows the programmer to "catch" errors ( exceptions ) before they are passed to the Ada runtime system, which will generally halt a program.

- In other words, exception handling allows the programmer to gracefully recover from errors ( notice that I didn't call them "bugs").

- In order to accomplish this the programmer must supply the necessary statements to handle the exceptions. A set of statements, proceeded by the reserved words **exception**, are called the **exception handler**.

Example :

```
    exception
        when Constraint_Error =>
            Text_IO.Put ( Item => "The input value is out of range ");
            Text_IO.New_Line;
        when Data_Error =>
            Text_IO.Put( Item => "The input value is
                                   not well formed");
            Text_IO.New_Line;
```

## Exceptions

If we anticipate that an exception may occur in a part of our program then we can write an exception handler to deal with it.

```
begin
        -- sequence of statements
exception
        when  Constraint_Error =>
        -- do something
end;
```

## Exceptions

```
begin
        -- sequence of statements
exception
        when  Constraint_Error =>
                Text_IO.Put ( " Numeric or Constrained error occurred");
        when Storage_Error =>
                Text_IO.Put ( " Ran out of space");
        when  Data_Error | My_Seq_IO.Data_Error =>
                Text_IO.Put ( " The input value is not well formed ");
        when others =>
                Text_IO.Put ( " Something else went wrong");
end;
```

- When an exception occurs control passes to the handler for the particular exception.
- If there is no handler, the subprogram terminates and the exception is passed back to the calling subprogram.
- If the calling subprogram has no handler, the exception is passed back to it's calling subprogram etc.
- When The statements within the handler have been executed, the program continues with the next statement after the block statement.

**Exceptions**

- Package Text_IO contains the predefined following exceptions :

  1. **Constraint_Error** - when something is out of range or a numeric operation cannot give a correct result.

  2. **Program_Error** - when we attempt to violate the control structure in some way.

  3. **Storage_Error** - accessible memory used up by too many recurcive calls.

  4 **Tasking_Error** - parallel programs communication failure.

- When the program executes if we break a language rule an exception may raised. When an exception occurs, the normal execution of the program ceases and if there is no exception handler the program terminates with an error message.

**Exceptions**

➢ The library package IO_Exceptions define the following exceptions needed by the predefined input - output packages :

1. **Status_Error** - attempt to read or write to a file not opened, or to open a file that is already open.

2. **Mode_Error** - attempt to read a Out_File or write to an In_File.

3. **Name_Error** - external file name can not be found.

4. **Use_Error** - attempt to open a file for illegal use. (Write to the keyboard).

5. **Device_Error** - failure of input or output device.

6. **End_Error** - attempt to read past the End of File.

7. **Data_Error** - attempt to read the wrong type.

8. **Layout_Error** - attempt to Set_Col or Set_Line that exceeds the maximum limits.

**Exceptions Continue**

➢ Also there is a library package named Ada.Exceptions which contatins the following main procedures and functions to handle all the exceptions.

1. procedure Raise_Exception ( E : in Exception_Id; Message : in String := "");

2. function Exception_Message ( X : Exception_Occerrence) return String;

3. procedure Reraise_Occurence ( X : Exception_Occurence);

4. function Exception_Identity ( X : Exception_Occurrence) return Exception_Id;

5. function Exception_Name ( X : Exception_Occurrence) return String;

4. function Exception_Information ( X : Exception_Occurrence) return String;

**Introduction to Exception Handling**

➢ Let's look at a complete program with exception handling statements included

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure RobustSumFact is

-- Prompts the user for an integer from 1 to 10
-- and displays the sum and factorial of all integers from  1 to N Sum
-- and Factorial are gotten from the functions sum and

        subtype OneToTen is Positive range 1..10;
        MaxNum      : OneToTen; -- input a value from one to ten
        SumToCount  : Positive; -- output - sum of integers from one to Count
        ProdToCount : Positive; -- output - product of integers from one to Count

package Integer_IO is new Integer_Io( Num => Positive);

        function sum( N : Positive) return Positive  is
                S : Positive:=1;
        begin
                if  N=1  then
                        return 1;
                else
                return N+sum(N-1);
        end if;
    end sum;

    function factorial(N : Natural) return Positive is
        begin
            if N =0 then
                return  1;
            else
                return N * factorial(N-1);
            end if;
    end factorial;

begin

    Put( Item => "Please enter an integer from 1 to 10>");
    Integer_IO.Get( Item => MaxNum);
    New_Line;
    Put( Item => " N     Sum     Factorial");
    New_Line;   Put( Item => " .......................");
    New_Line;
    for Count in 1 .. MaxNum loop
        SumToCount := Sum( N=> Count);
        ProdToCount:= factorial(N => Count);
        Integer_IO.Put(Item=> Count, Width =>3);
        Integer_IO.Put(Item=> SumToCount, Width =>7);
        Integer_IO.Put(Item => ProdToCount,Width =>9);
        New_Line;
    end loop;
```

```
Exception
        when Constraint_Error =>
                Put(Item => " The input value is out of range");
                New_Line;
        when Data_Error =>
                Put(Item => " The input value is not well formed");
                New_Line;

end RobustSumFact;
```

## Control Structures - Conditional Loops

- Recall that the loop variable in a for loop be a descrete type which is either incremented or decremented. Additionally, the number of times to iteratively execute command statements must be known in advance.

- Knowing in advance the number of times a sequencce of statements must be executed is not always possible. For instance, the program may be reading its input from a file of unknown length, or the user of the program may desire to input several data items for processing in one sitting.

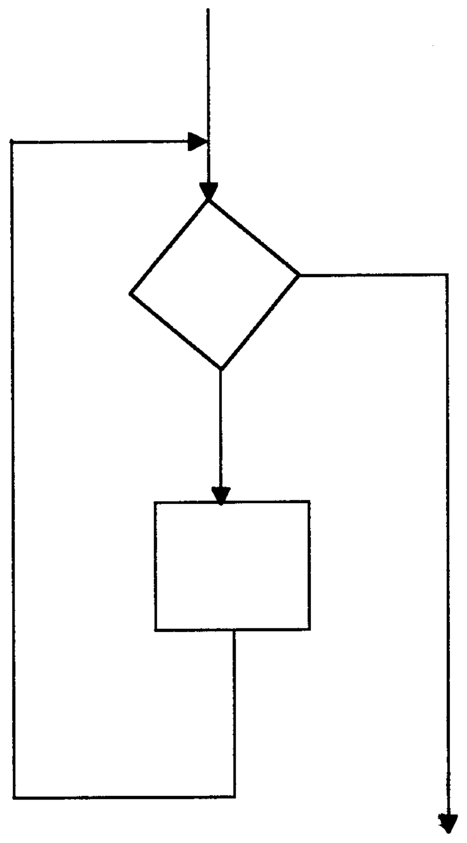
- This type of situation calls for something called a conditional loop. That is, it calls for a loop that will loop only while or until a certain condition exists ( e.g. haven't reached the last data item in the file yet).

- There are two ways in which this can be accomplished in Ada, through the use of (1) a loop with an **exit** statement or (2) the **while** loop construct.

**Control Statements**

- Iteration : the **loop** with **exit** statement

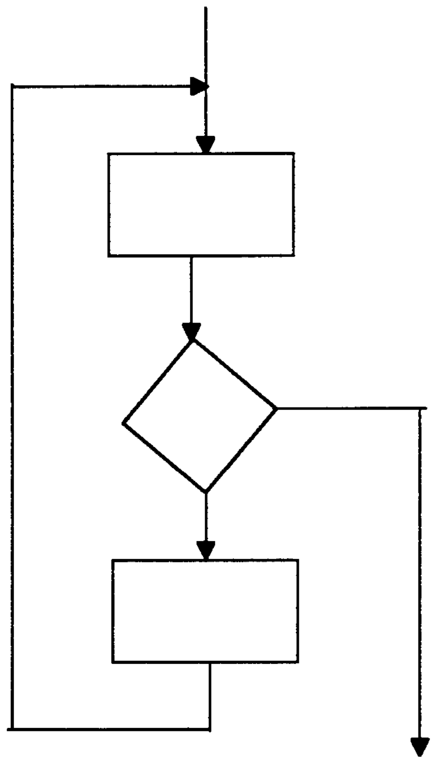
```
loop
        sequence_of_statements;
        exit when  Boolean_expression;
        sequence_of_statements;
end loop;
```



**Control Statements**

➢ Iteration : the loop statement with while

```
while Boolean_expression  loop
        sequence_of_statements;
end loop;
```



**Usage Example - while loop**

➢ Using the **for** loop we could only decrement or increment by the next whole unit. The example below shows how the **while** loop can be used to increment by 2 rather than by single whole units (1).

```
with Ada.Text_IO;
use  Ada.Text_IO;

procedure OddNumbers  is

-- Displays odd numbers from 1 to 39
        OddNumber : Integer;
package Integer_IO is new Integer_IO( Num => Integer);

begin  -- OddNumbers

        OddNumber := 1;
        while OddNumber <= 39 loop
                Integer_IO.Put( Item => OddNumber, Width => 3);
                OddNumber := OddNumber +2;
        end loop;
        New_Line;
end OddNumbers;
```

➢ Note the loop body is repeated "while" the condition ( OddNumber <+ 39) is true.

**While vs For loop**

➢ It is always possible to implement a **for** loop using a **while** loop; however, the converse is not true.

```
for  i  in 1..5  loop
        square := i * i ;
end loop;

i := 1;
while  i <=5  loop
        square := i * i;
        i := i + 1;
end loop;
```

➢ In the for loop the control variable i is declared implicitly and only exists within the body of the loop. That is, i is considered a local variable within the for block.

➢ In the while loop the variable i is just like any other variable and must be declared explicitly within the program. Its lifetime is like that any other "normal" variable.

➢ Additionally, i is explicitly incremented by 1 in the while loop; whereas, in the for loop it is implicitly incremented.

➢ In this example the for loop is more appropriate.

**While loop Control**

➢ Never Test for an exact value when using a Float

```
i := 0 ;
Epsilon := 2.0 / 3.0;
while ( i < 150.0) loop
        i := i + Epsilon;
end loop;
```

➢ Use of a Flag Variable

```
MoreData := 'Y';
while ( MoreData = 'Y')  loop
        Text_IO.Put ( "Do you have more data Y or N ? >");
        Text_IO.Get(MoreData);
end loop;
```

➢ Use of a Sentinel

```
Sentinel := 1;
Text_IO.Put ( "Enter score or enter -1 to quit");
Integer_IO.Get(Score);
while  ( Score /= Sentinel) loop
        Sum := Sum + Score;
        Text_IO.Put("Enter score or enter -1 to quit");
        Integer_IO.Get ( Score );
        Text_IO.New_Line;
end loop;
```

## Loop and Exit Statement

- Often it is useful to create a loop using the general loop construct with an exit statement.

- This may be particularily useful if we wish to execute a sequence of statements at least once before terminating a loop ( e.g. a main program loop which asks the user if he / she wishes to continue or quit).

```
loop
        -- Display Main Menu
        Text_IO.Get ( Response );
        exit when ( Response = 'q');
end loop;
```

- Notice also, a **while** loop can be implemented using the **exit when** loop construct but that the conditional logic is the opposite.

```
i := 0 ;
Epsilon := 2.0 / 3.0;
while ( i < 150.0) loop
        i := i + Epsilon;
end loop;


i := 0 ;
Epsilon := 2.0 / 3.0;
loop
        exit when (i < 150.0)
        i := i + Epsilon;
end loop;
```

## Robust Exception Handling

➢ The general loop with a simple **exit** can be used for input error checking. That's why subtyping comes in handy!!

```
loop
        begin
                -- Prompt the user for an input value.
        exit;   -- if valid data input
        exception -- if bad data input
                -- Determine which exception was raised
                -- notify user and take corrective action
end loop;
```

1. If bad data is input an exception is raised.

2. Control is passed to the exception handler.

3. After the exception handler has executed, control flows to the **end loop**.

4. The loop is repeated.

➢ In general, an exception hendler returns control to the end of the enclosing **begin-end** block in which it exists.

## Procedures

```
procedure   Procedure_Name ( formal_parameter_list) is
        declarative_part

begin
        statement_1;
        statement_2;
        statement_3;
        .
        .
        .
        statement_N;
end Procedure_Name;
```

- **Procedure Call**

  ```
  procedure_Name ( actual_parameter_list);
  ```

- Procedure differs from a function in that it does not return a result but a sequence of statements is put into actions.

## Procedures

- A procedure call is considered to be a statement.

- The actual parameter's types must be the same as the corresponding formal parameter's types.

- **In** actual parameter can be variable, constant or an expression. It must have a value.

- **In** formal parameter is considered to be a constant that is initialized at the time of the call. In the procedure it is not permitted to change the value of a in formal parameter.

- **Out** actual parameter must be variable.

- **Out** formal parameter is treated as a variable without an explicit initial expression. The value of an out parameter may be read, so we can use it.

- **In Out** actual parameter must be a variable. It must have a value.

- In Out formal parameter can be used as an ordinary variable. Its value can both be used and changed. If the values is changed the value of the actual parameter is changed.

- Variables declared within the procedure exist only within the procedure ( Local variables).

**Example**

```
with Ada.Text_IO;
use  Ada.Text_IO;

procedure Sample is

        N: Integer;
        M: Integer:=3;
package Integer_IO is new Integer_IO( Num => Integer);

        procedure Cube ( X: in Integer; Y: out Integer) is
        begin
                Y:= X**3;
        end Cube;

begin
        Cube( X=> 2, Y=> N);
        Integer_IO.Put(Item => N);
        New_Line;
        Cube( X => M, Y => N);
        Integer_IO.Put( Item => N);
        New_Line;
        Cube(X => M*2, Y => N);
        Integer_IO.Put( Item => N);
        New_Line;
end  Sample;
```

**Example**

```
with Ada.Text_IO;
use  Ada.Text_IO;

procedure Sample is
        N: Integer:=3;
package Integer_IO is new Integer_IO( Num => Integer);

        procedure Cube ( X: in out Integer) is
        begin
                X := X**3;
        end Cube;

begin
        Cube( X=> N);
        Integer_IO.Put(Item => N);
        New_Line;
end  Sample;
```

**Example**

```
with Ada.Text_IO;
use  Ada.Text_IO;

procedure Sample is

        N: Integer;
        M: Integer:=3;
package Integer_IO is new Integer_IO( Num => Integer);
        procedure Cube ( X: in Integer := 1; Y: out Integer) is
        begin
                Y:= X * X * X;
        end Cube;
begin
        Cube( X=> 2, Y=> N);
        Integer_IO.Put(Item => N);
        New_Line;
        Cube( X => 2, Y => N);
        Integer_IO.Put( Item => N);
        New_Line;
        Cube( M, Y => N);
        Integer_IO.Put( Item => N);
        Cube( Y => N);
        Integer_IO.Put( Item => N);
        New_Line;
end  Sample;
```

**Larger Example - A Simple Sort**

```
with Ada.Text_IO;
use  Ada.Text_IO;
procedure Sort3Numbers is
-- Reads three numbers and sorts them
-- so that they are in increasing order
        Num1 : Float;  -- a list of three cells
        Num2 : Float;
        Num3 : Float;
-- a procedure specification
        procedure Order ( X : in out Float; Y : in out Float);
-- a procedure body
        procedure Order( X : in out Float;  Y : in out Float) is
        -- Orders a pair of numbers represented by X and Y so that
        -- smaller numbers is in X and the larger number is in Y
        -- Pre X and Y  are assigned values
        -- Post X has the smaller value and Y has the larger value

                Temp : Float;
        begin
                if  X > Y   then
                        Temp := X;
                        X:=Y;
                        Y:=Temp;
                end if;
        end Order;
```

```
package Float_IO is new Float_IO( Num => Float);

begin
        Put (Item => " Enter e float numbers to be sorted. one per line");
        New_Line;
        Float_IO.Get( Item => Num1);
        Float_IO.Get( Item => Num2);
        Float_IO.Get( Item => Num3);

        Order ( X => Num1, Y => Num2); -- Order the data in Nm1 and Num2
        Order ( X => Num1, Y => Num3);  -- Order the data in Nm1 and Num3
        Order ( X => Num2, Y => Num3);  -- Order the data in Nm2 and Num3
        -- Display the results
        Put(Item => " The three numbers in order are");
        Float_IO.Put( Item => Num1, Fore=>5,Aft=>2,Exp=>0);
        Float_IO.Put( Item => Num2, Fore=>5,Aft=>2,Exp=>0);
        Float_IO.Put( Item => Num3, Fore=>5,Aft=>2,Exp=>0);
        New_Line;

end Sort3Numbers;
```

**Procedures in a Package - the Specification**

```
-- Package for getting numeric input robustly

package RobustInput is

        -- Gets an integer value in the range MinVal , MaxVal from the terminal
        -- Pre MinVal and MinVal are defined
        -- Post MinVal <= Item <= MaxVal

        procedure Get ( Item  : out Integer;
                        MinVal: in Integer;
                        MaxVal: in Integer);
        -- Gets a float value in the range MinVal, MaxVal from the terminal
        -- Pre MinVal and MinVal are defined
        -- Post MinVal <= Item <= MaxVal

        procedure Get ( Item  : out Float;
                        MinVal: in Float;
                        MaxVal: in Float);
end RobustInput;
```

**Procedures in a Package - the Body**

```
with Ada.Text_IO;
use Ada.Text_IO;
package body RobustInput is

package Float_IO is new Float_IO ( Num => Float);
package Integer_IO is new Integer_IO ( Num => Integer);

        procedure Get( Item  : out Integer;
                          MinVal: in Integer;
                          MaxVal: in Integer) is
        -- Gets an integer value in the range MinVal , MaxVal from the terminal
        -- Pre MinVal and MinVal are defined
        -- Post MinVal <= Item <= MaxVal

        subtype TempType is Integer range MinVal .. MaxVal;
        TempItem: TempType;          -- temporary copy of MinVal

        begin

                loop
                        begin   -- exception handler block
                        Put( Item => " Enter an Integer between ");
                        Integer_IO.Put( Item => MinVal, Width =>0);
                        Put(Item => "and");
                Integer_IO.Put( Item => MaxVal, Width =>0);
                Put( Item => ">");
                Integer_IO.Get( Item => TempItem);
                Item := TempItem;
                exit;  -- valid data
            Exception  -- invalid data
                when Constraint_Error =>
                Put(" Value entered is out of range. Please try again");
                New_Line;
                Skip_Line;
                when Data_Error =>
                Put(" value entered not an integer. Please try again");
                New_Line;
                Skip_Line;
            end; -- exception handler block
        end loop;
end Get;

procedure Get ( Item : out Float;
                MinVal: in Float;
                MaxVal: in Float) is
-- Gets a float value in the range MinVal, MaxVal from the terminal
-- Pre MinVal and MinVal are defined
-- Post MinVal <= Item <= MaxVal
        subtype TempType is float range MinVal .. MaxVal;
        TempItem : TempType;
begin
```

```
                    loop
                              begin -- exception handler block
                                        Put( Item => " Enter a floating
                                                            point value between");
                                        Float_IO.Put( Item => MinVal,
                                                            Fore => 1,Aft=>2,Exp=>0);
                                        Put( Item => " and ");
                                        Float_IO.Put( Item => maxVal,
                                                            Fore=>1,Aft=>2,Exp=>0);
                                        Put( Item => ">");
                                        Float_IO.Get( Item => TempItem);
                                        Item := TempItem;
                              exit;   -- valid data
                    exception   -- invalid data
                              when Constraint_Error =>
                              Put( Item => " Value is out of range. Please try again");
                              New_Line;
                              Skip_Line;
                              when Data_Error =>
                              Put( Item => " Value entered not floating point.
                                                  Please try again");
                              New_Line;
                              Skip_Line;
                    end; -- exception handler block
                    end loop;
          end Get;
end RobustInput;
```

**Main Procedure**

```
with RobustInput;
procedure TestRobustInput is

        subtype SmallInt is Integer range -10 .. 10;
        subtype LargerInt is Integer range -100 .. 100;
        subtype SmallFloat is Float range -10.0 .. 10.0;
        subtype LargerFloat is Float range -100.0 .. 100.0;

        Small : SmallInt;
        SmallF : SmallFloat;
        Larger : LargerInt;
        LargerF: LargerFloat;
begin
        Robustinput.Get(Small,SmallInt'First,SmallInt'Last);
        Robustinput.Get(Larger,LargerInt'First,LargerInt'Last);
        Robustinput.Get(SmallF,SmallFloat'First,SmallFloat'Last);
        Robustinput.Get(LargerF,LargerFloat'First,LargerFloat'Last);
end TestRobustinput;
```

## Numeric Data Types

➢ So far we have seen several predefined numeric data type ( subtypes) : Integer, Positive, Natural and Float. But why so many different types? Why not use a Float type for all our numeric operations?

1. We would like to try and use the most appropriate type for representing the values in a program.

2. Integer values require less storage.

3. Integer operations are faster than floating point operations.

➢ **Conversions among Numeric Types**

1. Ada does not allow mixing types in an expression; rather it requires explicit conversion.

```
T ( expression )
```

T is the name of a new converted numeric type .

Example:

```
Float_Number    : Float := 6.0;
Natural_Number : Natural := 3;

Float_Number    : = Float (Natural_Number ) ;
Natural_Number : = Natural ( Float_Number);
```

## Using an External Ada.Numerics.Elementary_Functions Library

➢ Many of the mathematical functions which we frequently used are not a part of the standard Ada language. However, most compilers supply a package to do these functions ( e.g.. Gnat's mathematics library is a child package Numerics.Elementary_Functions ).

➢ Also there is a another generic package Numerics.Generic_Elementary_Functions that we have to instantiate for numeric types.

➢ Because both Numerics.Elementary_Functions and Numerics.Generic_Elementary_Functions are children of the Numerics we must not use the clause **use if** we would like to have visibility in package Numerics.

Example:

```
with Ada.Text_IO;

use Ada.Text_IO;

with Ada.Numerics.Elementary_Functions;


procedure SquareRoots is

-- Illustrates the square root function provided by Numerics.Elementary_Functions

        MaxNumber   : constant   : Positive := 20;

package Integer_IO is new Integer_IO ( Num => Integer);

package Float_IO is new Float_IO ( Num => Float);


begin

        Put ( Item => "Number Square Root");

        New_Line;

        Put ( Item => "---------  -------------");

        New_Line;
```

```
    for Number  in  1..MaxNumber   loop
        Integer_IO.Put ( Item => Number, Width => 3);
        Float_IO.Put ( Item => Numerics.Elementary_Functions .
            Sqrt( Float(Number)), Fore => 7, Aft => 5, Exp => 0);
        New_Line;
    end loop;
end SquareRoots;
```

**The use clause**

- So far, when we have had occasion to take advantage of functions or procedures in external packages we have used the **with** clause and have prefixed the name of the package to the function or procedure. Ada provides a method to avoid this qualification in the form of the **use** clause.
- Be careful when the external package is a child and we use the use clause we can not have access to its parent package. For example if we write use Numerics.Elementary_Functions then we do not have access to the package Numerics ( parent of Elementary_Functions ) and on Ada ( which is the parent of Numerics).

Example:

```
with Ada.Text_IO;

use Ada.Text_IO;

with Ada.Numerics.Elementary_Functions;

use  Ada.Numerics.Elementary_Functions;


procedure SquareRoots is

-- Illustrates the square root function provided by Numerics.Elementary_Functions

        MaxNumber   : constant   : Positive := 20;

package Integer_IO is new Integer_IO ( Num => Integer);

package Float_IO is new Float_IO ( Num => Float);


begin

        Put ( Item => "Number Square Root");

        New_Line;

        Put ( Item => "---------  --------------");

        New_Line;
        for Number  in  1..MaxNumber  loop
                Integer_IO.Put ( Item => Number, Width => 3);
                Float_IO.Put ( Item => Sqrt( Float(Number)), Fore => 7, Aft => 5,
                                                                Exp => 0);
                New_Line;
        end loop;
end SquareRoots;
```

➢ The same SquareRoots procedure if we use the Generic_Elementary_Functios package.

```
with Ada.Text_IO;
use Ada.Text_IO;
with Ada.Numerics.Generic_Elementary_Functions;

procedure SquareRoots is
-- Illustrates the square root function provided by Numerics.Elementary_Functions
        MaxNumber   : constant   : Positive := 20;
package Integer_IO is new Integer_IO ( Num => Integer);
package Float_IO is new Float_IO ( Num => Float);
package my_math is new Generic_Elementary_Functions ( Float);
use my_math;
begin
        Put ( Item => "Number Square Root");
        New_Line;
        Put ( Item => "---------  --------------");
        New_Line;
        for Number  in  1..MaxNumber   loop
            Integer_IO.Put ( Item => Number, Width => 3);
            Float_IO.Put ( Item => Sqrt( Float(Number)), Fore => 7, Aft => 5,
                                                            Exp => 0);
            New_Line;
        end loop;
    end SquareRoots;
```

## Writing Mathematical Formulas in Ada

➢ The following examples below illustrate how simple equations are written in Ada.

| Mathematical Formula | Ada Expression |
|---|---|
| $b^2 - 4ac$ | B ** 2 - 4.0 * A * C |
| $a + b - c$ | A + B -C |
| $\frac{a+b}{c+d}$ | (A + B) / ( C + D) |
| $\frac{1}{1+a^2}$ | 1.0 / ( 1.0 + A ** 2) |
| $a \times -(b + c)$ | A * ( - ( B + C )) |
| $x^j$ | X ** J |

**Short - Circuit Boolean Operators**

- In addition to the Boolean operators which we have already examined, there are two additional operators **and then** and **or else**.

- Circumstances do arise when it is desirable to evaluate the right side of an **and** only if the left side is true, or the right side of an **or** if the left side is false. Especially if the evaluation process is a time consuming operation ( such as a computationally expensive function call) or if evaluation causes side effects ( like advancing a file pointer).

**or vs or else**

In the following example both sides of the expression are evaluated :

Flag **or** (( Y + Z) /= ( X - Z ))

but in the expression below the right side is only evaluated if Flag is false:

Flag **or else** (( Y + Z) /= ( X - Z ))

## Short - Circuit Boolean Operators

### and vs and then

Consider the expression below :

( X /= 0.0 ) **and** ( Y / X > 5.0)

If X is 0, the expression is false and therefore the entire expression would evaluate to false. Not only that, but if we tried to evaluate the right hand side we would generate a Constraint_Error.

If we rewrite our expression using the short - circuit and operator then we avoid that possible problem.

( X /= 0.0 ) **and then** ( Y / X > 5.0)

**The Character Type**

Earlier we discussed the Ada's character type, but now we will take a little closer look through the use of examples.

Recall that the set of all characters used is really a enumeration type which has been declared in the package Standard; therefore all of the operators which are valid for enumeration types can be used with Ada's Character type ( e.g. 'Val, 'Pos, In, etc. ).

Let's look at some of the features of the predefined character set used most frequently ( in our case ASCII ).

1. The digits are an increasing sequence of consecutive characters.

   '0' < '1' < '2' < . . . < '9'

2. The uppercase letters are an increasing set of consecutive characters.

   'A' < 'B' < 'C' < 'D' < . . . < 'Z'

3. The lowercase letters are an increasing set of consecutive characters.

   'a' < 'b' < 'c' < . . . < 'z'

4. The digit characters precede the uppercase characters and the uppercase characters precede the lowercase characters.

   '0' < '1' . . . < '9' < 'A' < 'B' < . . . < 'Z' < 'a' < 'b' . . . < 'z'

**Example**

Now let's look at a simple program that counts the number of blanks between words.

```
with Text_IO;
procedure BlankCount is
-- Counts the number of blanks in a sentence.
        Blank :    constant Character := ' ';  -- character being counted
        Sentinel : constant Character := '.';  -- sentinel character
        Next  : Character;          -- next character in sentence
        Count : Natural;            -- number of blank characters
package My_Int_IO is new Integer_IO ( Num => Integer);

begin  -- BlankCount
        Count :=  0;                -- Initialize Count
        Text_IO.Put(Item => "Enter a sentence ending with a period.");
        Text_IO.New_Line;

          -- Process each input character up to the period
        Text_IO.Get(Item => Next);      -- Get first character
        Text_IO.Put(Item => Next);
        while Next /= Sentinel loop   -- invariant: Count is the count of blanks so
                                      -- far and no prior value of Next is the sentinel
                if Next = Blank then
                        Count := Count + 1;        -- Increment blank count
                end if;
        Text_IO.Get(Item => Next);    -- Get next character
        Text_IO.Put(Item => Next);

    end loop;  -- assert: Count is the count of blanks and Next is the sentinel
    Text_IO.New_Line;
    Text_IO.Put(Item => "The number of blanks is ");
    My_Int_IO.Put(Item => Count, Width => 1);
    Text_IO.New_Line;

end BlankCount;
```

**Example**

Now let's look at a simple procedure that reads a character token and converts it to a Natural number.

```
with Text_IO;
procedure GetNaturalToken (NumData : OUT Natural) is
-- Reads consecutive characters ending with the symbol %.  Computes
-- the integer value of the digit characters, ignoring non-digits.
-- Pre:  None-- Post: NumData is the value of the digit characters read.
        Base :     constant Positive := 10;   -- the number system base
        Sentinel : constant Character := '%'; -- the sentinel character
        TempNum :  Natural;      -- to compute the numerical value
        Next :     Character;  -- each character read
        Digit :    Natural;                    -- the value of each numeric character
                                               -- (its ASCII position)

begin -- GetNaturalToken
-- Accumulate the numeric value of the digits in TempNum
        TempNum := 0;                      -- initial value is zero
        Text_IO.Get(Item => Next);              -- Read first character
        while Next /= Sentinel loop
        -- invariant:
        --   No prior value of Next is the sentinel and
        --   if Next is a digit, TempNum is multiplied by Base and
        --   Next's digit value is added to TempNum
        if (Next >= '0') and (Next <= '9') then
             -- Process digit
            Digit := Character'Pos(Next) - Character'Pos('0');  -- Get digit value
            TempNum := Base * TempNum + Digit; -- Add digit value
        end if;
        Text_IO.Get(Item => Next);          -- Read next character
        end  loop;
        -- assert:
        --   Next is the sentinel and
        --   TempNum is the number in base Base formed from the digit
        --   characters read as data

        NumData := TempNum;

    end GetNaturalToken;
```

**Putting it All Together - Loops, Case, Characters, Numbers and Packages**

➢ The following case study presents the opportunity to tie together many of the things we have discussed so far in the course.

The Case Study illustrates the following :

1. Using the **if** and **case** constructs.

2. Using **loops**.

3. Using **procedures** ( in packages ).

4. **Hidden procedures** within a package.

5. Using **subtypes** in a package.

6. Using existing **packages**.

7. Creating our own package.

8. General problem solving.

➢ The general idea of the case study is to develop a package which can be used to display the value of a number in words ( like you would on a check ).

**The Package Specification**

```
package NumToWord is
-- procedure to print a positive integer in words. The subtype
-- Natural16 is provided so that the package will be correct
-- on all Ada compilers including those for which Natural is 16 bits.
        subtype Natural16 is Natural RANGE 0 .. 32767;
        procedure PutInWords (Item: Natural16);
end NumToWord;
```

**The Package Body**

```
with Text_IO;
package body NumToWord is
        -- package body for displaying nonnegative integers in words
        subtype Digit IS Natural range 0..9;
        -- local procedures

        procedure Put1Digit (Item : Digit) is
        -- Puts its argument in words.
        -- Pre:  Item is assigned a value between 0 and 9.
        -- Post: Item is displayed in words.

        begin -- Put1Digit
                case Item is
                        when 0 => Text_IO.Put (Item => "zero");
                        when 1 => Text_IO.Put (Item => "one");
                        when 2 => Text_IO.Put (Item => "two");
                        when 3 => Text_IO.Put (Item => "three");
                        when 4 => Text_IO.Put (Item => "four");
                        when 5 => Text_IO.Put (Item => "five");
                        when 6 => Text_IO.Put (Item => "six");
                        when 7 => Text_IO.Put (Item => "seven");
                        when 8 => Text_IO.Put (Item => "eight");
                        when 9 => Text_IO.Put (Item => "nine");
                end case;
        end Put1Digit;
```

**The Package Body - Continued**

```
procedure Call1Digit (Units : Digit) is
-- Calls procedure Put1Digit with parameter Units if
-- Units is not zero.
-- Pre:  Units is assigned a value between 0 and 9.
-- Post: Calls Put1Digit if Units is not 0.

begin -- Call1Digit

    if Units /= 0 then
        Text_IO.Put(Item => " ");
        Put1Digit (Item => Units);
    end if;
end Call1Digit;
```

**The Package Body - Continued**

```
procedure Put2Digits (Item : Natural16) is
-- Puts its argument in words.
-- Pre:  Item is assigned a value between 0 and 99.
-- Post: Item is displayed in words.
-- Uses: Put1Digit
          Tens :  Digit;                      -- tens digit
          Units : Digit;                      -- units digit
begin -- Put2Digits
          Tens  := Item / 10;             -- Get tens digit
          Units := Item REM 10;             -- Get units digit
          case Tens is
                    when 0 =>
                              Put1Digit (Units);             -- less than ten
                    when 1 =>
                    case Units is                   -- in the teens
                              when 0 => Text_IO.Put (Item => "ten");
                              when 1 => Text_IO.Put (Item => "eleven");
                              when 2 => Text_IO.Put (Item => "twelve");
                              when 3 => Text_IO.Put (Item => "thirteen");
                              when 5 => Text_IO.Put (Item => "fifteen");
                              when 8 => Text_IO.Put (Item => "eighteen");
                              when 4 | 6 | 7 | 9 =>
                              Put1Digit (Units);  -- Put  ...teen
                              Text_IO.Put (Item => "teen");
                    end case;
```

**The Package Body - Continued**

```
            when 2 =>
                    Text_IO.Put (Item => "twenty");  -- Put twenty ...
                    Call1Digit (Units);
            when 3 =>
                    Text_IO.Put (Item => "thirty");   -- Put thirty ...
                    Call1Digit (Units);
            when 4 =>
                    Text_IO.Put (Item => "forty");     -- Put forty ...
                    Call1Digit (Units);
            when 5 =>
                    Text_IO.Put (Item => "fifty");        -- Put fifty ...
                    Call1Digit (Units);
            when 8 =>
                    Text_IO.Put (Item => "eighty");      -- Put eighty ...
                    Call1Digit (Units);
            when 6 | 7 | 9 =>
                    Put1Digit (Tens);      -- Put ...ty ...
                    Text_IO.Put (Item => "ty");
                    Call1Digit (Units);
            end case;
      end Put2Digits;
```

**The Package Body - Continued**

```
        procedure PutInWords (Item: Natural16) is
                CopyOfItem: Natural;
                Thousands:  Natural;
                Hundreds:   Natural;
        begin
                if Item = 0 then
                        Put1Digit (Item => Item);
                else
                        CopyOfItem := Item;
                        if CopyOfItem >= 1000 then
                                Thousands := CopyOfItem / 1000;
                                Put2Digits (Item => Thousands);
                                Text_IO.Put (Item => " thousand ");
                                CopyOfItem := CopyOfItem REM 1000;
                        end if;
                        if CopyOfItem >= 100 then
                                Hundreds := CopyOfItem / 100;
                                Put1Digit (Item => Hundreds);
                                Text_IO.Put (Item => " hundred ");
                                CopyOfItem := CopyOfItem REM 100;
                        end if;
                        if CopyOfItem /= 0 then
                                Put2Digits (Item => CopyOfItem);
                        end if;
                end if;
        end PutInWords;
end NumToWord;
```

**The Test Program**

```
with Text_IO;
with RobustInput;
with NumToWord;
procedure TestNumToWord is

-- program to demonstrate and test NumToWord.PutInWords
        N: NumToWord.Natural16;
package My_int_IO is new Integer_IO ( Num => Integer);

begin -- TestNumToWord
        for Count IN 1..5 loop
                Text_IO.Put (Item => "Integer ");
                My_Int_IO.Put (Item => Count, Width => 1);
                Text_IO.New_Line;
                RobustInput.Get (Item => N,
                                   MinVal => NumToWord.Natural16'First,
                                   MaxVal => NumToWord.Natural16'Last);
                NumToWord.PutInWords (Item => N);
                Text_IO.New_Line;
        end loop;
end TestNumToWord;
```

## Composite Types : Records and Arrays

- Composite types give us a way to group data items into larger "chunks". This is analogous to grouping statements together in blocks, subprograms, and packages.

- Ada provides the record and array type constructors, which can be used to form composite types from simpler types.

- A record is a data structure containing a group of related data items; the individual components, or fields, of a record can contain data of different types. We can use a record to store a variety of information about a person, such as the person's name, marital status, age, date of birth, and so on. Each data items is stored in a separate record field; we can reference each data item stored in a record through its field name. For example, Person.Name references the field Name of the record Person.

- Also a record can be extended with additional components by using tagged type record. The new record inherits exactly the structure, operations, values and also all the additional components from the parent record.

- An array is a data structure used for storage of a collection of data items that are all of the same type (e.g. all the exam scores for a class). Using an array allows us to associate a single variable name ( e.g. Scores) with the entire collection of data. This enables us to save the entire collection of data in main memory ( one item per memory cell) and to reference individual items easily. For example, the third score in the array Scores would be referenced as Scores(3).

## A Record Type for Employee Records

Given the following declarations :

```
NameSize  : constant Positive  := 20;

subtype IDRange is Positive Range 1111..9999;
subtype NameType is String ( 1..NameSize);
type GenderType is (Female, Male);
type DepartmentManager is (Research, Administration, Headquarters);
```

➢ We declare a record type Employee:

```
type Employee is
    record
        ID : IDRange;
        Name : NameType;
        Gender : GenderType;
        NumDepend : Natural;
        Rate : NonNegFloat;
        TaxSal : NonNegFloat;
    end record;
```

**Record Type Continue**

➢ We declare a record type Employee as tagged type:

```
type  Employee  is tagged
        record
                ID : IDRange;
                Name  : NameType;
                Gender  : GenderType;
                NumDepend  : Natural;
                Rate  : NonNegFloat;
                TaxSal  : NonNegFloat;
        end record;
```

Now we can extend the Employee record to a new one Manager record
( more details later in the object oriented paragramming with Ada 94)

```
type  Manager  is  new  Employee with
        record
                DeptManager  : DeptType;
        end record;
```

The Manager records inherits all the fields , operations from the Employee ( parent) and an additional field named DeptManager.

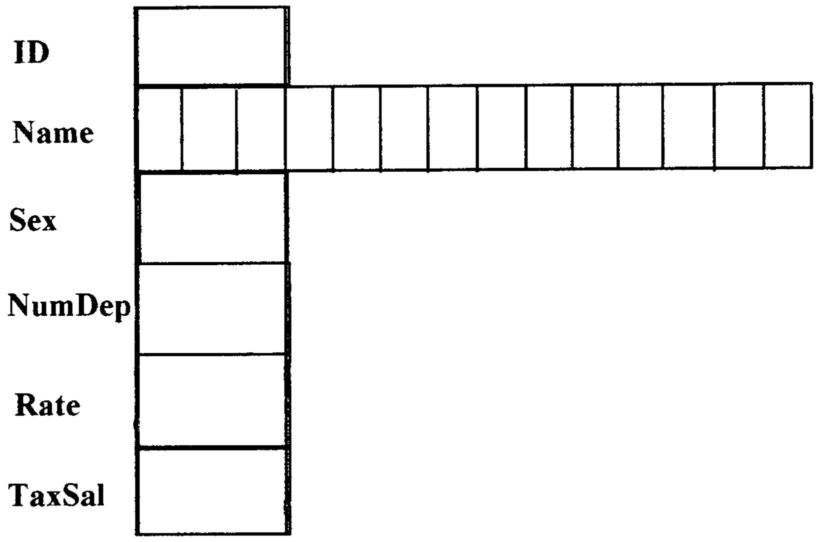
➢ A mathematical type, perhaps used in graphics :

```
type  Point  is
        record
                X  : Float;
                Y  : Float;
        end record;
```

**Record Variables or Objects**

The record type definnition, like all other type definition, creates no object or variable, allocates no memory. It is a template that describes the format of each record and the name of each individual data element. A variable declaration is required to allocate storage space record. The record variables Clerk and Janitor are declared next.

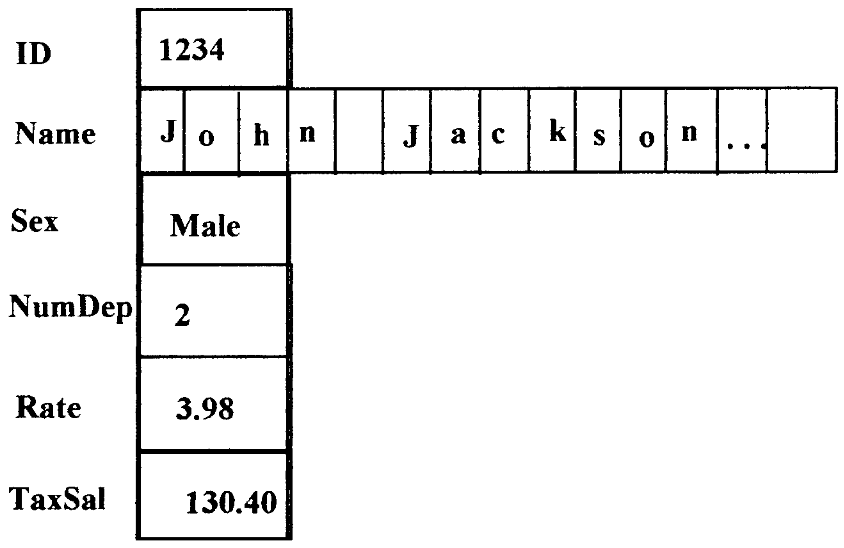
Clerk   : Employee;
Janitor  : Employee;



## Working with Record Objects

Data can be stored in Clerk through this sequence of assignment statements :

```
Clrerk. ID  := 1234;
Clerk.Name  := " John Jackson ";
Clerk. Gender  := Male;
Clerk. NumDepend  := 2;
Clerk. Rate  := 3.98;
Clerk. TaxSal  := Clerk>Rate * 40.0 - Float ( Clerk.NumDepend) * 14.40;
```



**Working with Record Objects ( Continue)**

Data can also be stored in Clerk through the use af a aggregate assignment of the form :

```
Clerk :=        (ID => 1234,
                 Name => "John Jackson",
                 Gender => Male,
                 NumDepend => 2,
                 Rate => 3.98,
                 TaxSal => 130.40);
```

➢ Once data are stored in a record, they can be manipulated in the same way as other data in memory.

The statements

```
Text_IO.Put ( Item => "The clerk is");
case Clerk.Gender is
        when Female  => Text_IO.Put( Item => :Ms. ");
        when Male    = > Text_IO.Put( Item => :Mr. ");
end case;
Text_IO.Put ( Item => Clerk.Name );
```

Displays the clerk's name after an appropriate title (Ms., or Mr.)

```
The clerk is Mr. John Jackson
```

**Operations on Record Objects**

Remember : A type is a set of values and a set of operations

**Set of values**:

A record value is a collection of related data values of different types. Each data value is stored in a separate field of the record.

**Set of oparations:**

- **Store**: if R1 is a record or tagged record ( parent)with a field named F1 and expression E is compatible with F1, then

```
R1.F1 := E;
```

stores the result of evaluating E in field F1 of record R1.

**Store**: if R1 is an extended tagged record ( child) with a field named F1 and expression E is compatible with F1, then

```
R1'( F => E)
```

stores the result of evaluating E in field F1 of record R1.

- **Retrieve**: If the field R1.F1 is compatible with variable C then

```
C := R1.F1;
```

retrieves the value in field F1 of record R1 and copies it into C. We can also write

```
R1.F1 := R2.F1;
```

## Operations on Record Objects - Continue

➢ **Assignment :** If R1 and R2 are record variables of the same type, the statement

R1 := R2;

copies all values associated with record R2 to record R1.

➢ **Comparison** : The result of the Boolean expression

R1 = R2

is true if and only if each of the fields of R1 is equal to its corresponding field in R2;

R1 /+ R2

is true if and only if at least one of the fields of R1 is not equal to its corresponding field in R2.

## Records Objects - Another Example

```
with Text_IO;
with Ada.Numerics.Elementary_Functions;
use Ada.Numerics.Elementary_Functions;;

procedure DistOrigin is
-- Finds the distance from a point to the origin.
        type Point is
                record
                        X : Float;
                        Y : Float;
                end record;

        Point1 : Point;            -- the data point
        Distance : Float;          -- its distance to the origin
package My_Flt_IO is new Float_IO ( Num => Float);

begin -- DistOrigin

        Text_IO.Put(Item => "Enter X coordinate (floating point) > ");
        My_Flt_IO.Get(Item => Point1.X);
        Text_IO.Put(Item => "Enter Y coordinate (floating point) > ");
        My_Flt_IO.Get(Item => Point1.Y);
        Distance := sqrt(Point1.X ** 2 + Point1.Y ** 2);
        Text_IO.Put(Item => "Distance to origin is ");
        My_Flt_IO.Put(Item => Distance, Fore=>1,Aft=>2,Exp=>0);
        Text_IO.New_Line;

end  DistOrigin;
```

## Records as Parameters

Records can be passed to subprograms, as parameters of any mode, or returned as the result of a function, just as simple objects can.

```
procedure GetPoint ( Item : out Point ) is

-- reads the X and Y coordinates of a pilot
-- Pre : none
-- Post : the point. item, is defined with values from the user

begin

    Text_IO.Put ( Item => "Enter X coordinate ( floating point)   , ");
    My_Flt_IO.Get ( Item => Item.X);
    Text_IO.Put ( Item => "Enter Y coordinate ( floating point)   , ");
    My_Flt_IO.Get ( Item => Item.Y);

end GetPoint;
```

**Hierarchical Records**

A hierarchical record is one that has a record as one or more of its fields.

```
StringLength  : constant  Positive := 20;
ZipCodeLength  : constant  Positive := 5;
subtype  IdRange  is Positive  range 1111..9999;
subtype  EmpString  is String ( 1.. StringLength);
subtype  ZipString  is String ( 1.. ZipCodeLength);
type  GenderType  is  ( Female, Male );

type  Employee  is
        record
                ID : IDRange;
                Name  : EmpString;
                Gender  : GenderType;
                NumDepend  : Natural;
                Rate  : NonNegFloat;
                TaxSal  : NonNegFloat;
        end record;
type  Address  is
        record
                Street   : EmpString;
                City     : EmpString;
                State    : EmpString;
                ZipCode: EmpString;
        end record;
```

➢ Finally, here is the declaration of the hierarchical record for an employee, and a declaration of an employee variable. Notice how simple the employee record is, given the component record types; notice also how we have used the Date type provided by the package Dates developed in the previous section.

```
type  NewEmployee   is
          record
                    PayData    : Employee;
                    Home       : Address;
                    StartDate  : Dates.Date;
                    BirthDate  : Dates.Date;
          end record;
Programmer   : NewEmployee;
```

The field selector

Programmer.StartDate;

references the subrecord StartDate ( type Date ) of the variable Programmer.

The field selector

Programmer.StartDate.Year

references the Year field of the subrecord Programmer.StartDate.

The field selector

Programmer.Year

is incomplete ( which Year field?) and would cause a syntax error.

The record copy statement

```
Programmer.StartDate  := DayOfYear;
```

copies each field of DayOfYear into the corresponding field of the subrecord Programmer.StartDate.

The statements

```
Text_IO.Put ( Item => "Year started);
My_Int_IO.Put( Item => Programmer.StartDate.Year, Width => 4);
Text_IO.Put ( Item => "Month started);
My_Int_IO.Put( Item => Programmer.StartDate.Month, Width => 4);
```

display two fields of the subrecord Programmer.StartDate.

The statements

```
Text_IO.Put ( Item => Programmer.PayData.Name);
Text_IO.Put ( Item => " started work in  );
My_Int_IO.Put( Item => Programmer.StartDate.Year, Width => 4);
```

display the line

```
John Jackson     started work in 1985
```

The computation for taxable salary could be written as

```
Programmer.PayData.TaxSal  := Programmer.PayData.Rate *
                                   40.0 -
                                   Programmer.PayData.NumDepend *
                                   14.40;
```

**Hierarhical Records - Continued**

The procedure below is used to input data for an employee record.

```
procedure  ReadEmployee ( OneClerk  : out Emplyee ) is
-- Reads one employee record into OneClerk
-- Pre : None
-- Post : Data are read into record OneClerk

begin -- ReadEmployee

        Text_IO.Put ( Item => "ID>");
        My_Int_IO.Get ( Item => OneClerk.ID);
        Text_IO.Put ( Item => "Name");
        Text_IO.Get ( Item => OneClerk.Name);
        Text_IO.Put ( Item => "Gender ( Female or Male)>");
        GenderType_IO.Get ( Item => OneClerk.Gender);
        Text_IO.Put ( Item => "Number of dependents>");
        My_Int_IO.Get ( Item => OneClerk.NumDepend);
        Text_IO.Put ( Item => "Hourly rate>");
        My_Flt_IO.Get ( Item => OneClerk.Rate);

end ReadEmployee;
```

**Hierarhical Records - Continued**

The procedure below is used to input data for an address record.

```
procedure ReadAddress ( Add : out Address ) is
-- Reads the fields of one address record

begin -- ReadAddress

    Text_IO.Put ( Item => " Please enter 20 character street address >");
    Text_IO.Get ( Item => Add.Street);
    Text_IO.Put ( Item => " Please enter 20 character city name >");
    Text_IO.Get ( Item => Add.City);
    Text_IO.Put ( Item => " Please enter 20 character state name >");
    Text_IO.Get ( Item => Add.State);
    Text_IO.Put ( Item => " Please enter 5 character ZipCode >");
    Text_IO.Get ( Item => Add.ZipCode);

end ReadAddress;
```

**Hierarhical Records - Continued**

```
procedure ReadNewEmp ( NewEmp : out NewEmployee) is

-- Reads a record into record variable NewEmp
-- Uses procedures ReadEmployee, ReadAddress and Dates.Get.
-- Dates.Get is a simple package which allows us to represent calendar dates
-- in a form convinient for reading.

begin -- ReadNewEmp

        ReadEmployee ( NewEmp.PayData);
        ReadAddress ( NewEmp.Home);
        SimpleDates.Get (NewEmp.StartDate);
        SimpleDates.Get ( NewEmp.BirthDate);

end ReadNewEmp;
```

**Arrays**

An array is a data structure used for storage of a collection of items that are all the same type; that is, all the elements of an array are homogeneous.

Array Type Declaration

**type** array_type **is array** subscript_type **of** element_type;

- ➢ The identifier array_type describes a collection of array elements; each of which can store an item of type element_type.

- ➢ The subscript_type can be discrete type (either predefined or user-defined). it may uses arbitrary universal expressions for each bound (e.g. -1..3+5)

- ➢ The element_type can be any predefined or user-defined scalar or composite type.

**Example Declaration :**

The following declaration

```
type Float_Array  is array (1..8)  of Float; -- define the type
X : Float_Array    -- create a variable
```

Creates a collection of eight memory cells with the name X in main memory ( usually contiguous allocation ) each element of which can hold exactly one Float value.

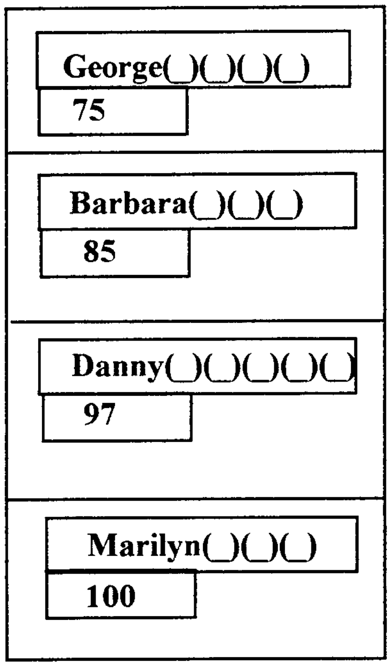
| X(1) | X(2) | X(3) | X(4) | X(5) | X(6) | X(7) | X(8) |
|---|---|---|---|---|---|---|---|
| 16.0 | 12.0 | 6.0 | 8.0 | 2.8 | 12.0 | 14.0 | -56.0 |

**Another Example**

The following declaration

```
type Student_Array  is array (1..4)  of Student; -- define the type
Student :  Student_Array    -- create a variable
my_Student : array ( 1..4) of student; -- create a variable with array type definition
```

Creates a collection of eight memory cells with the name Student in main memory ( usually contiguous allocation ) each element of which can hold exactly one Student record.



**More Example Declarations :**

The following four groups of statements all produce the same result: namely they create an array Scores which can hold ten Integers.

1. Scores : **array** ( 1..10) **of** Integer; -- create a variable

**2. type** Integer_Array **is array** (1..10) **of** Integer; --define a type
Scores : Integer_Array; --create avariable

**3. subtype** Index **is** Integer **range** 1..10; --define a subscript
**type** Integer_Array **is array** ( Index)) **of** Integer; --define a type
Scores : Integer_Array; --create avariable

4. Scores : **Array** ( Index) of Integer; --create a variable

**Array Type Differentation**

A, B : array(1..10) of Integer; --A and B are the same type

A : array(1..10) of Integer; --Note A and B are no longer considered
B : array(1..10) of Integer; --to be the same type!!

**Elementary Array Operations**

```
Old_Scores , New_Scores, Sum  : Integer_Array;     --creates three array variables
                                                   -- each of which can hold
                                                   -- then Integers


New_Scores(3) := 25;           -- assigns the third element of the array New_Scores
                               -- the value 25


New_Scores(3) := (25 * 3) +15;      -- assigns the third element of the
                                    -- array New_Scores the value of the
                                    -- result of the expression
                                    -- (25 * 3) + 15


New_Scores (6) := Old_Scores (3);   -- assigns the value contained in the third
                                    -- element of the array Old_Scores to the
                                    -- sixth element of the array New_Scores


Sum(1) := Old_Scores(1) + New_Scores (1);       -- stores the sum of the first
                                                -- element of Old_Scores and
                                                -- New_Scores in the first
                                                --element of the array Sum


New_Scores := Old_Scores;  -- since both arrays are the same size (and type) this
                           -- statement copies the contents of the array
                           -- Old_Scores to the array New_Scores.
```

```
My_Int_IO.Put ( Item => Old_Scores (5), Width => 5);     -- displays the fifth
                                                         -- element of the array
                                                         -- Old_Scores

My_Int_IO.Get ( Item => New_Scores (2);  -- Takes the value provided by the
                                         -- user and places the value in the
                                         -- second element of the array
                                         -- New_Scores.
```

**Array Element Processing**

```
type Float_Array  is Array (1..8)  of  Float;
X : Float_Array := (16.0, 12.0, 6.0,, 8.0, 2.5, 12.0, 14.0, -54.5);
i : Integer := 6 ;

My_Flt_IO.Put( X(4));          --displays the value 8.0
My_Flt_IO.Put( X(i));          --displays the value 12.0
My_Flt_IO.Put( X(i) + 1.0);   --displays the value 13.0
My_Flt_IO.Put( X(i +1 ));     --displays the value 14.0
My_Flt_IO.Put( X(i + i));      --Illegal index (12) out of range
My_Flt_IO.Put( X(2 * i - 4)); --displays the value -54.5


X ( i ) := X( i+1 );     -- assigns value of X(7) to X(6)
X ( i )  = X ( i ) +1.0  -- adds 1.0 to the contants of X (6)
X ( i - 1 ) := X ( i );  -- assigns value X(6) to X(5)
X ( i ) -1 := X ( i );   -- Illegal assignment
```

**Operations of Array Types**

A'First : Denotes the lower bound of the first index range. Its type is the corresponding index type.

```
type Table is array ( 1..100 ) of Integer;
Table'First = 1
```

A'First (N) : Denotes the lower bound of the N-th. Its type is the corresponding index type.

```
type Matrix is array (1..20, 1..30);
Matri'First(1) =1
```

A'Last : Denotes the upper bound of the first index range. Its type is the corresponding index type.

```
type Table is array ( 1..100 ) of Integer;
Table'Last = 100
```

A'Last (N) : Denotes the upper bound of the N-th. Its type is the corresponding index type.

```
type Matrix is array (1..20, 1..30);
Matri'Last(1) =20
Matrix'Last(2) = 30
```

A'Range is equivalent to the range A'First .. A'Last except that the prefix A is only evaluated once.

A'Range(n) is equivalent to the range A'First(N) .. A'Last(N) except that the prefix A is only evaluated once.

A'Length : Denotes the number of values of the first index

Table'Length = 100

A'Length(N) : Denotes the number of values of the N-th index range

Matrix'Length(2) = 30

**Aggregate Array Assignment**

An array can be field to each filled with values by three methods :

- Assignment to each element using individual assoignment statements.
- Copying one array to another
- Using aggregate assignment

Using named association :

```
New_Scores := (1=> 80, 2=> 56, 3 => 78, 4 => 95);
```

or equivalently

```
New_Scores := (80, 56, 78,  95);
```

- We frequently would like to initialize ( or clear ) an array, this can be done in several different ways :

```
for i in 1..10 loop
        New_Scores(i) := 0;
end loop;
```

or equivalently

```
New_Scores := (1..10 => 0);
```

**Aggregate Array Assignment**

- Suppose we wished to initialize the first 4 elements as above but all the rest to 0:

```
New_Scores := ( 1=> 80, 2 => 56, 3 => 78, 4 => 95, others =>0);
```

- We wish to initialize the first, third and fifth element to a non-zero value and all elements to 0:

```
New_Scores := ( 1=> 80,  3 => 78, 5 => 85, others =>0);
```

- Finally, we know realize that we can initialize all elements of our array New_Scores to 0 with the simple statement :

```
New_Scores := ( others =>0);
```

**Subscript Examples**

```
Max_Elements : constant Positive := 8;
subtype Index is Positive range 1..Max_Elements;
type Float_Array is array (Index) of Float;
X : Float_Array;


Seats_Per_Row : constant Positive := 50;
type Seating_Status_Array is array ( 1.. Seats_Per_Row ) of Boolean;
Row_1, Row_2, Row_3 : Seating_Status_Array;
Row_1(50 := true;


Min_Temp : constant Integer := -50;
Max_Temp : constant Integer := 150;
type Temp_Frequency_Array is array ( Min_Temp .. Max_Temp ) of Natural;
Temp_Frequence : Temp_Frequency_Array ;


subtype Name_Length is Positive range 1..15;
type Name_Array is array ( Name_Length) of Character;
First_Name, Last_Name : Name_Array;
First_Name := ( 'I', 'c', 'a', 'b', 'o', 'd', others => '');
Last_Name := ( 'C', 'r', 'a', 'n', 'e', thers =>'');
```

**Subscript Examples**

```
subtype Lower_Case is Character range 'a' .. 'z';
type Letter_Frequency_Array is array ( Lower_Case ) of Natural ;
Letter_Frequency : Letter_Frequency_Array;
Letter_Frequency := 73;


type Gender is (male, female);
type Gender_Frequency_Array is array (Gender) of Natural;
Gender_Frequency : Gender_Frequency_Array;
Gender_Frequency(male) := 149;


subtype Plain_Letters is Character range 'A' .. 'Z';
type Cipher_Alphabet is array ( Plain_Letters ) of Character ;
Cipher_Code : Cipher_Alphabet := ( 'F', 'G', 'V', 'X', 'B');
Text_IO.Put ( Item => Cipher_Code ( D ));


type Car_Maker is ( Chrysler, Ford, General_Motors );
subtype Dollars is Float range 0.00 .. Float'Last;
Car_Sales := array ( Car_Maker ) of Dollars;
Car_Sales ( Ford) := 23_000_000.00;
My_Flt_IO.Put ( Item => Car_Sales( Ford ));
```

## Using for Loops with Arrays

We can use a for loop to step through every element of an array for processing.

```
Size : constant Positive := 10;
subtype Index is Positive range 1..Size;
type Integer_Array is array ( Index ) of Integer;
Squared : Integer_Array;



for i in 1..Size loop
        Square(i) := i * i ;
end loop;

Max_Students : constant Positive := 30;
subtype Class is Integer range 1..Max_Students;
type Test_Score_Array is array ( Class) of Integer;
Test1, Test2, Total_Score : Test_Score_Array;

for Student_No in Class'First .. Class'Last loop
        Total_Score ( Student_No ) := Test1(Student_No) + Test2(Student_No);
end loop;
```

## Arrays as Parameters

- Arrays can be passed as a parameter to a function or a procedure just like any scalar type we have seen.

- Recall that there are three modes for parameters : **in**, **out** and **in out**.

- An array passed as an actual parameter to a formal **in** parameter is always copied into the formal parameter. This can obviously take time and memory.

- Recall also, that a scalar **out** and **in out** parameters are copied back into the calling program just before the subprogram terminates.

- However, for efficiency purposes, Ada allows **out** and **in out** parameters of composite types to be passed by reference ( that is, copying an address of the actual parameter rather than the entire contents).

- Although the method is more efficient, it can create some problems if the subprogram terminates abnormally because some of the elements of the actual parameters may have been changed!

**Case Study 8.7**

**Problem**

You want a program that keeps track of your monthly expenses in each of several categories. The program should read each expense amount, add it to the appropriate category total, and print the total expenditure by category. The input data consists of the category number and amount of each purchase made during the past month.

**Analysis**

You have selected these budget categories: entertainment, food, clothing, rent, tuition, insurance, and miscellaneous. Seven separate totals are to be accumulated; each total can be associated with a different element of a seven element array. The program must read each expenditure, determine to which category it belongs, and then add that expenditure to the appropriate array element. When done with all expenditures, the program can print a table showing each category and its accumulated total.

**Data Type**

```
Type  Categories  is (Entertainment, Food, Clothing, Rent, Tuition,
                      Insurance, Miscellaneous)
        Problem Inputs
                each expenditure and its category

        Problem Outputs
                the array of seven expenditure totals
```

**Case Study 8.7**

**Algorithm ( a jazz song written by our vice president).**

1. Initialize all category totals to zero.

2. Read each expenditure and add its total to the appropriate category.

3. Display the accumulated total for each category.

**Case Study 8.7 - The Program**

```
with Text_IO;
with Sreen;
procedure HomeBudget is
-- Prints a summary of all expenses by budget category.
    MaxExpense :  constant Float := 10_000.00;    -- max expense amount
    type Categories is (Entertainment, Food, Clothing, Rent,
                        Tuition, Insurance, Miscellaneous);
    type Commands is (E, F, C, R, T, I, M, Q);

    package Category_IO is new Text_IO.Enumeration_IO(Enum =>
                                                      Categories);
    package Command_IO is new Text_IO.Enumeration_IO(Enum =>
                                                      Commands);
    package My_Flt_IO is new Text_IO.Enumeration_IO(Num => Float);
    subtype Expenses is Float range 0.00..MaxExpense;     -- expense type
    type BudgetArray is array (Categories) of Expenses;  -- array type
    Budget : BudgetArray;              -- array of ten totals

    procedure Initialize (Budget : out BudgetArray) is
    -- Initializes array Budget to all zeros.
    -- Pre:  None
    -- Post: Each array element Budget(Category) is 0.00
    begin  -- Initialize
        Budget := (OTHERS => 0.00);
    end Initialize;
```

```
procedure DisplayTitles is
-- displays a list of expense categories with their abbreviations
        WhichRow: Screen.Depth;
begin
        Text_IO.New_Page;
        Text_IO.New_Line;
        Text_IO.New_Line;
        Text_Io.New_Line;
        Text_IO.Put(Item => "                Expense Categories");
        Text_IO.New_Line;
        Text_IO.New_Line;
        WhichRow := 5;
        for C in Commands'First..Commands'Pred(Commands'Last) loop
                Screen.MoveCursor ( Row => WhichRow, Column => 20);
                Command_IO.Put(Item => C, Width => 3);
                Category_IO.Put(Item=>
                                        Categories'Val(Commands'Pos(C)));
                WhichRow := WhichRow + 1;
        end loop;
        Screen.MoveCursor(Row => WhichRow, Column => 20);
        Command_IO.Put(Item => Commands'Last, Width => 3);
        Text_IO.Put(Item => "when data entry is completed");
end DisplayTitles;
```

```
procedure GetCommand(Command: out Commands) is
-- Reads a category command from the terminal
-- Post:  a valid Command is returned
begin
    loop
        begin   -- exception handler block
            Screen.MoveCursor(Row => 18, Column => 15);
            Text_IO.Put("Please enter first letter
                        of category > ");
            Command_IO.Get(Item => Command);
            Screen.MoveCursor(Row => 19, Column => 15);
            Text_IO.Put("Category accepted, thank you");
        exit;
        exception
            when Text_IO.Data_Error =>
                Screen.Beep;
                Screen.MoveCursor(Row => 19,
                                    Column => 15);
                Text_IO.Put("Sorry,invalid category!    ");
                Text_IO.Skip_Line;



        end;    -- exception handler block
    end loop;   -- assert: valid command input received
end GetCommand;
```

```
procedure GetExpense(Expense: out Expenses) is
-- Reads an expense from the terminal
-- Post:  a valid Expense is returned
begin
        loop
                begin  -- exception handler block
                        Screen.MoveCursor(Row => 20, Column => 15);
                        Text_IO.Put("Please enter expense as
                                        floating point number > ");
                        My_Flt_IO.Get(Item => Expense);
                        Screen.MoveCursor(Row => 21, Column => 15);
                        Text_IO.Put("Expense accepted, thank you");
                        exit;
                exception
                        when Text_IO.Data_Error =>
                                Screen.Beep;
                                Screen.MoveCursor(Row => 21,
                                                        Column => 15);
                                Text_IO.Put("Sorry, invalid expense!    ");
                                Text_IO.Skip_Line;
                end;    -- exception handler block
        end loop;    -- assert: valid expense received
end GetExpense;
```

```
    procedure Post (Budget : in out BudgetArray) is
-- Reads each expenditure amount and adds it to the appropriate
-- element of array Budget.
-- Pre:  Each array element Budget(c) is 0.0
-- Post: Each array element Budget(c) is the sum of expense
-- amounts for category c.

        Sentinel : constant Commands := Q; -- sentinel command
        NextCommand :  Commands;         -- command
        NextCategory : Categories;       -- expenditure category
        NextExpense :  Expenses;         -- expenditure amount
    begin  -- Post
        loop
        -- invariant:
        -- no prior value of NextCommand  is Sentinel
            GetCommand(Command => NextCommand);
            exit when NextCommand = Sentinel;
            NextCategory:=   Categories'Val
                        (Commands'Pos(NextCommand));
            GetExpense(Expense => NextExpense);
            Budget(NextCategory) := Budget(NextCategory) +
                                    NextExpense;
        end loop;
    end Post;
```

```
procedure Report (Budget : IN BudgetArray) is

-- Displays the expenditures in each budget category.
-- Pre:  Each array element Budget(c) is assigned a value.
-- Post: Each array element Budget(c) is displayed.
    WhichRow: Screen.Depth;

begin  -- Report
    Screen.ClearScreen;
    Screen.MoveCursor(Row => 3, Column => 20);
    Text_IO.Put(Item => "Category     Expense");
    Text_IO.New_Line;   Text_IO.New_Line;
    WhichRow := 5;

    for Category in Categories loop
        Screen.MoveCursor(Row => WhichRow, Column => 20);
        Category_IO.Put(Item => Category, Width=>13);
        My_Flt_IO.Put(Item=>Budget(Category),
                              Fore=>7, Aft=>2, Exp=>0);
        WhichRow := WhichRow + 1;
    end loop;

    Screen.MoveCursor(Row => 23,Column => 1);
end Report;
```

```
begin -- HomeBudget
        -- prepare terminal screen for data entry
        DisplayTitles;
        -- Initialize array Budget to all zeros.
        Initialize (Budget);
        -- Read and process each expenditure.
        Post (Budget);
        -- Print the expenditures in each category.
        Report (Budget);
end HomeBudget;
```

## Strings in Ada

Ada's string type is actually defined in Standard as follos :

**type** String **is array** ( Positive **range** <>) **of** Character;

**type** Wide_String **is array** ( Positive **range** <>) **of** Wide_Character;

making strings just a special case of unconstrained arrays ( we'll see the more general case later).

- A string variable is in fact an array of characters, with a subscript range that must be a subtype of Positive.

- String variables can be compared and assigned like other Ada variables, but their lengths must match exactly.

- It is possible to assign or refer to a part, or slice, of a string.

- Strings can be concateneted, or "pasted together" to form longer ones.

- Package Ada.Strings and its child packages Maps, Fixed, Bounded, Constants, Unbounded contain predefined routines for String types.

- Package Ada.Strings also contains child packages Wide_Maps, Wide_Constants, Wide_Fixed, Wide_Bounded, Wide_Unbounded which handle predefined routines for Wide_String types.

## Declaring and Using String Variables

The declarations

```
NameSize  : constant Positive  := 11;
FirstName : String ( 1.. NameSize );
LastName : String ( 1.. NameSize );
```

allocate storage for two string variables : FirstName and LastName. String variables FirstName and LastName can store 11 characters each ( subscript range 1..11). In general, a string variable of type String ( 1..N) can be used to store a string of up to N characters.

```
FirstName := "Johnny";
```

will raise Constraint_Error but

```
FirstName := "Johnny     ";
FirstName(1..6):="Johnny";
```

will not raise Constriant_Error

**Declaring and Using String Variables**

Given the declarations

```
FirstNameLength : Natural;
LastNameLength : Natural;
```

the statements

```
Text_IO.Put ( Item => "Enter your first name followed by CR, ");
Text_IO.Get ( Item => FirstName, Last => FirstNameLength);
Text_IO.Put ( Item => "Enter your name name followed by CR, ");
Text_IO.Get ( Item => LastName, Last => LastNameLength);
```

can be used to enter string values into the string variables FirstName and LastName. Up to 11 characters can be stored in FirstName and LastName. If the data characters Johnny are entered after the first prompt and the data characters Appleseed are entered after the second prompt, string FirstName is defined as :

| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) | (11) |
|---|---|---|---|---|---|---|---|---|---|---|
| J | o | h | n | n | y | ? | ? | ? | ? | ? |

and string LastName is defined as

| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) | (11) |
|---|---|---|---|---|---|---|---|---|---|---|
| A | p | p | l | e | s | e | e | d | ? | ? |

The variables FirstNameLength and LastNameLength will contain 5 and 9, respectively.

**Declaring and Using String Variables**

The statement

```
        Text_IO.Put ( Item => FirstName ( 1.. FirstNameLength));
```

displays the string Johnny.

Given the declarations

```
        WholeNameLength  : Natural;
        WholeName        : String (1..24);
```

the statements

```
        WholeNameLength := FirstNameLength + LastNameLength +2;
        WholeName(1.. LastNameLength) := LastName(1..LastNameLength);
        WholeName(LastNameLength+1 .. LastNameLength + 2) := ", ";
        WholeName( LastNameLength+3 .. WholeNameLength ) :=
                        FirstName ( 1 .. FirstNameLength);
        Text_IO.Put ( Item => WholeName( 1.. WholeNameLength));
```

will store in WholeName, and display :

```
        Appleseed,  Johnny
```

## String Concatenation

The string concatenation operator &, applied to two strings S1 and S2, concatenetes, or "pastes together" its two arguments. The statement

```
S3 := S1 & S2;
```

stores in S3 the concatenation of S1 and S2. For the assignment to be valid, the length of S3 still must match the sum of the lengths of S1 and S2; if it does not, Constraint_Error will be raised, as usual.

WholeName can be created more simply using concatenetion:

```
WholeNameLength := FirstNameLength + LastNameLength +2;
WholeName(1.. WholeNameLength) := LastName(1..LastNameLength)
& ", " & FirstName ( 1 .. FirstNameLength);
```

The result of a concatenation can also be passed directly as a parameter, for example to Text_IO.Put :

```
Text_Io.Put ( Item => LastName(1..LastNameLength)
& ", " & FirstName ( 1 .. FirstNameLength));
```

Concatenation is actually defined for all 1-dimensional unconstrained array types, but is most often used with strings.

**Case Study - Cryptogram Generator**

```
with Text_IO;
procedure Cryptogram is

    subtype Letter is Character range 'A'..'Z';
    type CodeArray is array (Letter) of Character;
    Code     : CodeArray;              -- array of code symbols

    function Cap (InChar : Character) return Character is

    -- returns an upper-case letter
    -- Pre: InChar is defined
    -- Post: if InChar is a lower-case letter, returns its upper-case
    --   equivalent; otherwise, returns InChar unmodified

    begin -- Cap

        if InChar in 'a'..'z' then
            return Character'Val(Character'Pos(InChar)
                - Character'Pos('a') + Character'Pos('A'));
        else
            return InChar;
        end if;
    end Cap;
```

```
procedure ReadCode (Code : out CodeArray) is

 -- Reads in the code symbol for each letter.
-- Pre : None
-- Post: 26 code symbols are read into array Code.

begin -- ReadCode

      Text_IO.Put(Item => "Enter a code symbol under each letter.");
      Text_IO.New_Line;
      Text_IO.Put(Item=>"ABCDEFGHIJKLMNOPQRSTUVWXYZ");
      Text_IO.New_Line;

      -- Read each code symbol into array Code.

      for NextLetter  in Letter loop
            Text_IO.Get(Item => Code(NextLetter));
      end loop;

      Text_IO.Skip_Line;

 end ReadCode;
```

```
procedure Encrypt (Code : CodeArray) is

-- Reads a plaintext (unencoded) message and displays its coded version.
-- Pre : The code for letter i is saved in Code(i).
-- Post: Displays the encoded message
        subtype Line is String(1..80);
        PlainText : Line;
        CodedText : Line;
        HowLong   : Natural;
begin -- Encrypt

        Text_IO.Put(Item => "Enter each character of your message.");
        Text_IO.New_Line;
        Text_IO.Put(Item => "No more than 80 characters, please.");
        Text_IO.New_Line;
        Text_IO.Put(Item => "Enter a CR after your message.");
        Text_IO.New_Line;
        Text_IO.Get_Line (Item => PlainText, Last => HowLong);
        for WhichChar in 1..HowLong loop
                if Cap(PlainText(WhichChar))  in Letter   then
                        CodedText(WhichChar):=
                        Code(Cap(PlainText(WhichChar)));
                else
                        CodedText(WhichChar) := PlainText(WhichChar);
                end if;
        end loop;
```

```
            Text_IO.Put (Item => CodedText(1..HowLong));
            Text_IO.New_Line;
        end Encrypt;

begin -- Cryptogram
        -- Read in the code symbol for each letter.
        ReadCode (Code);
        -- Read each character and print the cryptogram
        Encrypt (Code);
end Cryptogram;
```

## Searching an Array

### Algorithm

1. Start with the first array element.

2. **while** the current element does not match the target **and** the current element is not the last element **loop**.

3. Advance to the next element **end loop**;

4. **if** the current element matches the target **then return** its subscript **else** return zero.

**Searching an Array**

```
function Search (Scores: ScoreArray; ClassSize: ClassRange;
                        Target: ScoreRange) return ClassRange;
-- Searches for Target in array Scores
-- Pre : ClassSize and subarray Scores(1..ClassSize) are defined
-- Post: Returns the subscript of Target if found;
--    otherwise, returns 0
        CurrentScore: ClassIndex;       -- array subscript


begin  -- Search
-- Compare each value in Scores to Target until done
        CurrentScore := 1;          -- Start with the first record
        while (Scores(CurrentScore) <> Target) and
                                         (CurrentScore <= ClassSize) loop
        -- invariant:
        -- CurrentScore <= ClassSize + 1 and
        -- no prior array element was Target
                CurrentScore := CurrentScore + 1; -- advance to next score
        end loop;
-- assertion: Target is found or last element is reached.
-- Define the function result.
        if Scores(CurrentScore) = Target then
                return CurrentScore;
        else
        return 0;
        end if;
end  Search;
```

**Sorting an Array**

Algorithm for Selection Sort

```
1. for Position ToFill in reverse 2..N loop

2. Find the largest element in subarray 1..PositionToFill.

3. if the largest element is not at subscript PositionToFill then
Exchange the largest element with the one at subscript PositionToFill.
end if
end loop
```

**Sorting an Array**

```
with Text_IO;
procedure SortScores is
-- Test program for procedure SelectSort
-- Sorts and displays an array of test scores
-- The array is initialized with eleven non-zero scores.
        MaxSize : constant Positive := 250;
        MaxScore : constant Positive := 100;
        subtype ClassIndex is Positive range 1..MaxSize;
        subtype ClassRange is Natural  range 0..MaxSize;
        subtype ScoreRange is Natural  range 0..MaxScore;
        type ScoreArray is array (ClassIndex) of ScoreRange;
        Scores : ScoreArray;
        ClassSize : ClassRange;
        package My_Int_IO is new Integer_IO ( Num => Inteher);
        procedure Exchange(Score1, Score2: in out ScoreRange) is
         -- exchanges two values of type ScoreRange
        -- Pre: Score1 and Score2 are defined
        -- Post: the values of Score1 and Score2 are interchanged
                TempScore: ScoreRange;
        begin
                TempScore := Score1;
                Score1    := Score2;
                Score2    := TempScore;
        end Exchange;
```

```
procedure SelectSort(Scores: in out ScoreArray;
                     ClassSize: in ClassRange) is

    IndexOfMax: ClassRange;

begin

    for PositionToFill in reverse 2..ClassSize loop
    -- Find the element in subarray 1..PositionToFill
    -- with largest Score
        IndexOfMax := PositionToFill;
        for ItemToCompare in reverse 1..PositionToFill - 1 loop
         if Scores(ItemToCompare) > Scores(IndexOfMax) then
            IndexOfMax := ItemToCompare;
         end if;
        end loop;

    -- assert: element at IndexOfMax is largest in subarray
        if IndexOfMax /= PositionToFill then
          Exchange(Scores(PositionToFill),Scores(IndexOfMax));
        end if;
    end loop;

end SelectSort;
```

```
          procedure DisplayScores(Scores: ScoreArray; ClassSize: ClassRange) is
          begin
                    for I in 1..ClassSize loop
                              My_Int_IO.Put(Item => I, Width => 3);
                              Text_IO.Put(Item => "  ");
                              My_Int_IO.Put(Item => Scores(I), Width => 4);
                              Text_IO.New_Line;

                    end loop;
          end DisplayScores;

begin -- SortScores
          ClassSize := 11;
          Scores := (75, 25, 100, 62, 79, 80, 85, 75, 91, 67, 68, OTHERS => 0);
          Text_IO.Put(Item => "Original Test Array:");
          Text_IO.New_Line;
          Text_IO.New_Line;
          DisplayScores(Scores => Scores, ClassSize => ClassSize);
          Text_IO.New_Line;
          SelectSort(Scores => Scores, ClassSize => ClassSize);
          Text_IO.New_Line;
          Text_IO.Put(Item => "Sorted Test Array:");
          Text_IO.New_Line;  Text_IO.New_Line;
          DisplayScores(Scores => Scores, ClassSize => ClassSize);
          Text_IO.New_Line;
end SortScores;
```

**Case Study- Sorting an array of Records**

```
with Text_IO;
with SimpleDates;

procedure SortScoreFile is
-- Sorts and displays an array of test score records
-- The records are read from a file SCORES.DAT
    MaxSize : CONSTANT Positive := 250;
    MaxScore : CONSTANT Positive := 100;
    subtype StudentName is String(1..20);
    subtype ClassIndex is Positive range 1..MaxSize;
    subtype ClassRange is Natural  range 0..MaxSize;
    subtype ScoreRange is Natural  range 0..MaxScore;

    type ScoreRecord is
        record
            Name:  StudentName;
            Score: ScoreRange;
        end record;

    type ScoreArray is array (ClassIndex) of ScoreRecord;
    package My_Int_IO is new Integer_IO( Num => Integer);
    Scores : ScoreArray;
    ClassSize : ClassRange;
```

```
procedure GetRecords(Scores: out ScoreArray;

                ClassSize: out ClassRange) is
        TestScores: Text_IO.File_Type; -- program variable naming the
                                        --input file
        TempSize:  ClassRange;   TempRecord: ScoreRecord;
begin -- GetRecords

-- Open the file and associate it with the file variable name
        Text_IO.Open    (File => TestScores, Mode => Text_IO.In_File,
                            Name => "SCORES.DAT");
-- Read each data item
-- and store it in the appropriate element of Scores
        TempSize := 0;                  -- initial class size
 -- Read each array element until done.
        while (not Text_IO.End_Of_File(TestScores)) and
                (TempSize < MaxSize) loop
        -- invariant:
        --  Records remain in the file and
         --  TempSize <= MaxSize
                Text_IO.Get(File => TestScores,
                            Item => TempRecord.Name);
                My_Int_IO.Get(File => TestScores,
                            Item => TempRecord.Score);
                 TempSize := TempSize + 1;
                Scores(TempSize) := TempRecord;    -- Save the score
        end loop;
        -- assert:
        --   End of file reached or
        --   TempSize is MaxSize

        if TempSize = MaxSize then
                Text_IO.Put(Item => "Array is filled.");
                Text_IO.New_Line;
        end if;

        ClassSize := TempSize;

end GetRecords;



procedure Exchange(Student1, Student2: in out ScoreRecord) is

        TempRecord: ScoreRecord;

begin
        TempRecord := Student1;
        Student1 :=   Student2;
        Student2 :=   TempRecord;

end Exchange;
```

```
procedure SelectSort(Scores:   in out ScoreArray;
                     ClassSize: in ClassRange) is
    IndexOfMax: ClassRange;
begin

    for PositionToFill in reverse 2..ClassSize loop
    -- Find the element in subarray 1..PositionToFill with largest Score

        IndexOfMax := PositionToFill;
        for ItemToCompare in reverse
                        1..PositionToFill-1 loop
            if Scores(ItemToCompare).Score  >
              Scores(IndexOfMax).Score    then
                IndexOfMax := ItemToCompare;
            end if;
        end loop;
    -- assert: element at IndexOfMax is largest in subarry
        if IndexOfMax /= PositionToFill then
           Exchange(Scores(PositionToFill),Scores(IndexOfMax));
        end if;
     end loop;

end  SelectSort;
```

```
    procedure DisplayScores(Scores: ScoreArray; ClassSize: ClassRange) is
    begin
        for I in 1..ClassSize loop
            My_Int_IO.Put(Item => I, Width => 3);
            Text_IO.Put(Item => "  ");
            Text_IO.Put(Item => Scores(I).Name);
            My_Int_IO.Put(Item => Scores(I).Score, Width => 4);
            Text_IO.New_Line;
        end loop;
    end DisplayScores;
begin -- SortScoreFile
    Text_IO.Put(Item => "Today is ");
    SimpleDates.Put(Item => SimpleDates.Today);  Text_IO.New_Line;
    GetRecords(Scores => Scores, ClassSize => ClassSize);
    Text_IO.Put(Item => "Original Test File:");
    Text_IO.New_Line;
    Text_IO.New_Line;
    DisplayScores(Scores => Scores, ClassSize => ClassSize);
    SelectSort(Scores => Scores, ClassSize => ClassSize);
    Text_IO.New_Line;
    Text_IO.Put(Item => "Sorted Test File:");
    Text_IO.New_Line;
    Text_IO.New_Line;
    DisplayScores(Scores => Scores, ClassSize => ClassSize);
    Text_IO.New_Line;
end SortScoreFile;
```

**Problem Solving Using Abstraction**

The strategy of decomposing a problem into smaller sub-problems which are more manageable and thus more easily solved is known as the "divide and conquer strategy"

Procedural abstraction embodies the concept of "divide and conquer" by focusing on the abstract solution of smaller problems. This is, one separate out what is to be achieved by the procedure from the details of how the procedure actually works.

The abstraction notion of what a procedure is expected to do then serves as the specification for the implementation of the procedure.

We have already seen the stepwise refinement process for building a program, which in essence continually refines our notion of what a program is to until we actually develop the code that implements these notion.

A program then, as we have seen and experienced in our earlier projects, is best built incrementally. There are two primary strategies for accomplishing this (1) **Top-down design** and (2) **Bottom-up design**

## Problem Solving Using Abstraction

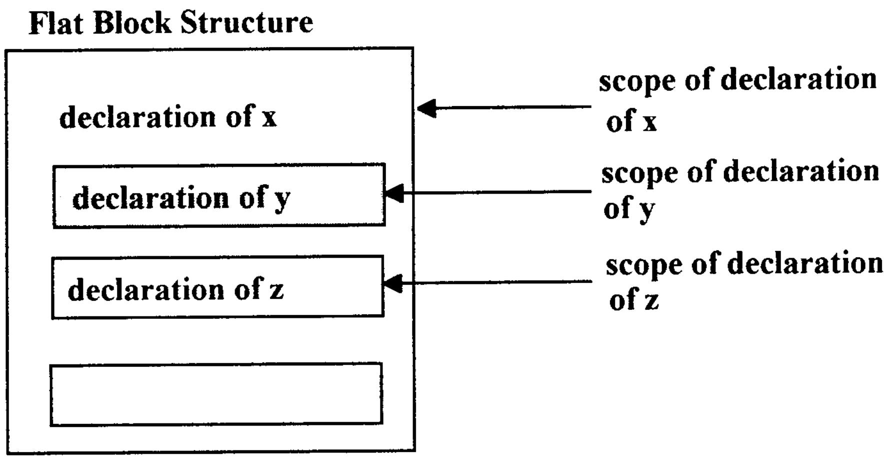
In top-down program development, you code at least a substantial part of the main program, and then test the overall flow using limited version of your procedures that are referred to as **program stubs.**

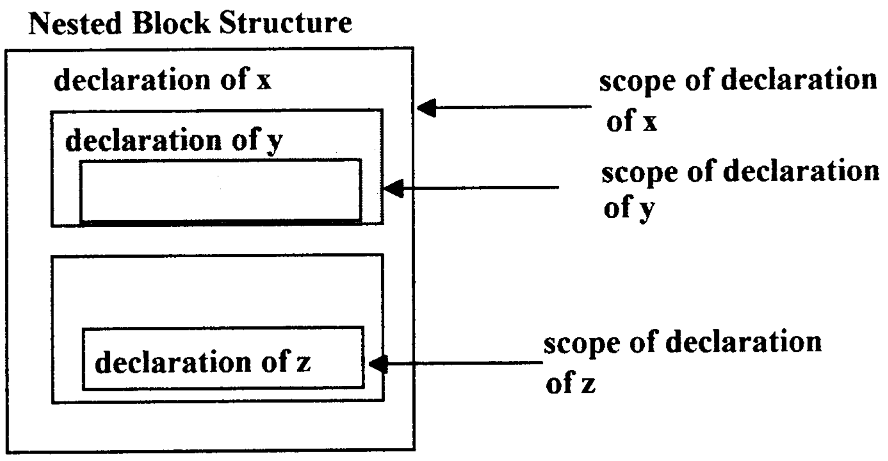
In the bottom up process, you write the procedure one at a time and test each one using a very simple "test harness" whose only purpose is to test and debug the procedure.

Generally, you will find that a programmer uses both strategies in development of large programs.

## Nested Procedures and Scoping Rules

- Ada is a block structured language that permits one subprogram to be nested within another

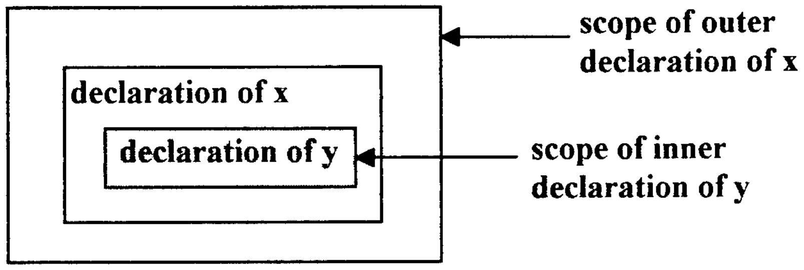

Flat Block Structure
declaration of x
declaration of y
declaration of z
scope of declaration of x
scope of declaration of y
scope of declaration of z


Nested Block Structure
declaration of x
declaration of y
declaration of z
scope of declaration of x
scope of declaration of y
scope of declaration of z

## Nested Procedures and Scoping Rules

- It is possible for the same identifier to be declared in different blocks.

- If the same identifier is declared in two nested blocks : the outer declaration is said to be **invisible** or to be **hidden** by the inner declaration.


declaration of x
declaration of y
scope of outer declaration of x
scope of inner declaration of y

## Nested Procedures and Scoping Rules

- Since all procedures are nested within the main program block, an identifier declared in the main program may be referenced anywhere in the program. Variables declared in the main program are referred to as **global variables.**

- It is generally not a good idea to reference global variables in the body of a subprogram because (1) it may produce undesirable side effects and (2) it, in effect couples the subprogram to the main program ( that is - it is counter to the concept of module independence).

- An identifier may be declared only once in a given procedure ; however, the same identifier may be declared in more than one procedure ( e.g. in the declaration area or in the formal parameter list).

- If an identifier is not declared locally, then a declaration in an outer block containing the point of reference is used.

Nested Procedures - Example

```
with Text_IO;

procedure TestTriangle is

    procedure Triangle (NumRows : IN Natural) is
    -- Prints a triangle by displaying lines of increasing length.
    -- The number of lines is determined by NumRows.
    -- Pre:  NumRows is assigned a value.
    -- Post: A triangle is displayed.
    -- Requirements: Calls procedure PrintLine to display each line.

        procedure PrintLine (NumStars : IN Natural) is
         --  Prints a row of asterisks.  The number of
         --  asterisks printed is determined by NumStars.
        --  Pre:  NumStars is assigned a value.
        --  Post: A row of asterisks is displayed.
            Star : CONSTANT Character := '*'; -- symbol being printed
        begin  -- PrintLine
        -- Print a row of asterisks
            for CountStar in 1 .. NumStars loop
                Text_IO.Put(Item => Star);
            end loop;
            Text_IO.New_Line;
        end PrintLine;
    begin  -- Triangle
    -- Print lines of increasing length
        for Row IN 1 .. NumRows loop
            PrintLine (NumStars => Row);
        end loop;
    end Triangle;

begin -- TestTriangle

    Triangle(NumRows => 6);

end TestTriangle;
```

**Text Files**

➢ A text file is a collection of characters stored under the same name on disk.

➢ In the Text_IO package a predefined type is provided namely :

**type** File_Type **is limited private**

➢ The end of each line of a text file is terminated by a special character called the end-of-line character <eol> ( Enter )

➢ The end of a text file is also terminated by a special character called the end of file character <eof> ( Ctrl-d)

➢ Although a text file may normally contain characters it can also store numerical data:

Example

```
1234   345 <eol>
999     -17<eol><eof>
```

**Text Files**

- ➢ Two useful functions provided by the Text_IO package are the End_of_Line function and the End_of_File function.

- ➢ Text_IO.End_of_Line (filename)- returns a value of true if the text character in the file is <eol>

  ( note : omission of the file name assumes the file referred to is

  Text_IO.Standard_Input).

- ➢ Text_IO.End_of_File ( filename)- returns a value of true if the next character in the file is <eof>. If an attempt is made to read the file again after the function returns true then an End_Error exception is raised.

  ( note : omission of the file name assumes the file referred to is

  Text_IO.Standard_Input).

- ➢ A text file must be declared just like any other variable in an Ada program :

```
InData  : Text_IO.File_Type;
OutData : Text_IO.File_Type;
```

**Text Files**

At any given time, a file can be used for either input, output or append but not simultaneously.

- If file is to be used for **input** the file must, of course, exist and to be opened for reading. The procedure call

```
          Text_IO.Open(File => InData, Mode => Txet_IO.In_File,
                                  Name =>"Scores.Dat");
```

prepares the file InData for reading by moving the file pointer to the beginning of the file and associating the variable identifier InData with the DOS file Score.Dat.

- If a file is to be used for output the file must be first created. That is the procedure call

```
          Text_IO.Create(File => OutData, Mode => Txet_IO.Out_File,
                                  Name =>"Test.Out");
```

prepares the file OutData for output. If a file by that name already exists it is deleted and a new one of the same name is created, otherwise a just a new empty file by that name is created.

- If we wish to add new text at the end of an existng file then we use the Append_File mode

```
          Text_IO.Open(File => InData, Mode => Txet_IO.Append_File,
                                  Name =>"Scores.Dat");
```

- Reading and writing a file users the same familiar get and put procedures that we are already accustomed to using : except a new parameter is added to the call.

Example

```
Text_IO.Get( File => InData, Item => NextCh) -- gets NextCh from the file
Text_IO.Put( File => OutData, Item => NextCh) -- writes NextCh to the file
```

**Sample Program**

```
with Text_IO;

procedure CopyFile is
-- program copies its input file TEST.DAT into its output file TEST.OUT
-- then closes TEST.OUT, re-opens it for input,
-- and displays its contents on the screen.
    InData  : Text_IO.File_Type;
    OutData : Text_IO.File_Type;
    NextCh  : Character;
begin -- CopyFile

    Text_IO.Open(File=>InData,Mode=>Text_IO.In_File,
                Name=>"TEST.DAT");
    Text_IO.Create(File=>OutData,Mode=>Text_IO.Out_File,
                Name=>"TEST.OUT");
    while not Text_IO.End_of_File(File => InData) loop
        while not Text_IO.End_of_Line(File => InData) loop
            Text_IO.Get(File => InData, Item => NextCh);
            Text_IO.Put(File => OutData, Item => NextCh);
        end loop;
        Text_IO.Skip_Line(File => InData);
        Text_IO.New_Line(File => OutData);
    end loop;
    Text_IO.Close(File => InData);
```

```
          Text_IO.Close(File => OutData);
          Text_IO.Open(File=>InData,Mode=>Text_IO.In_File,
                    Name=>"TEST.OUT");
          while not Text_IO.End_of_File(File => InData) loop
               while not Text_IO.End_of_Line(File => InData) loop
                    Text_IO.Get(File => InData, Item => NextCh);
                    Text_IO.Put(Item => NextCh);
               end loop;
               Text_IO.Skip_Line(File => InData);
               Text_IO.New_Line;
          end loop;

          Text_IO.Close(File => InData);
          exception
               when Text_IO.Name_Error =>
                    Text_IO.Put(Item => "File TEST.DAT doesn't exist in this
                                                              directory!");
                    Text_IO.New_Line;
     end CopyFile;
```

**Case Study**

```
with Text_IO;
with Screen;

procedure Histogram is

-- Plots a histogram on the screen consisting of vertical bars.
-- Each bar represents the frequency of occurrence of a given
-- alphabet letter in the input file.
-- The input file is assumed to be Standard_Input; use input redirection
-- if you wish to use a disk file instead.

      subtype UpperCase is Character range 'A'..'Z';
      subtype LowerCase is Character range 'a'..'z';
      type List is array(Character range <>) of integer;
      Uppers   : List(UpperCase);
      Lowers   : List(LowerCase);

      NextCh   : Character;
      Scale    : Natural;  MaxCount : Natural := 0;
      WhichCol : Screen.Width;

      package My_Int_IO is new Integer_IO( Num => Integer);
```

```
procedure Plot(WhichCol  : Screen.Width;
               BottomRow : Screen.Depth;
               HowMany   : Screen.Depth;
               WhichChar : Character) is

-- draws one vertical bar on the screen
-- Pre: WhichCol, BottomRow, HowMany, and WhichChar are defined
-- Post: draws a bar in column WhichCol, using character WhichChar
--      to do the plotting. The bottom of the bar is given by
--      BottomRow; the bar contains HowMany characters.

begin -- Plot

    for Count IN 0 .. Howmany - 1 loop
        Screen.MoveCursor(Column => WhichCol,
                          Row => BottomRow - Count);
        Text_IO.Put(Item => WhichChar);
    end loop;

end Plot;
```

```
begin -- Histogram
    -- initialize letter-counter arrays

    Uppers := (others => 0);
    Lowers := (others => 0);

    -- read each character in the file; update letter counters
    while not Text_IO.End_Of_File loop
        while not Text_IO.End_Of_Line loop
            Text_IO.Get(NextCh);
            case NextCh is
                when UpperCase =>
                    Uppers(NextCh) := Uppers(NextCh) + 1;
                    if Uppers(NextCh) > MaxCount then
                        MaxCount := Uppers(NextCh);
                    end if;
                when LowerCase =>
                    Lowers(NextCh) := Lowers(NextCh) + 1;
                    if Lowers(NextCh) > MaxCount then
                        MaxCount := Lowers(NextCh);
                    end if;
                when others =>
                    null;
            end case;

        end loop;
```

```
            Text_IO.Skip_Line;
        end loop;
        Scale := MaxCount / 20 + 1;
        Screen.ClearScreen;  Screen.MoveCursor(Row => 1, Column => 15);
        Text_IO.Put(Item => "Scale: 1 star = ");
        My_Int_IO.Put(Item => Scale, Width => 1);
        Text_IO.Put(Item => " occurrences");
        Screen.MoveCursor(Row => 22, Column => 4);
        Text_IO.Put(Item=>
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ");
        WhichCol := 4;

        for C in LowerCase loop
            if Lowers(c) /= 0 then
                Plot(WhichCol, 21, Lowers(C) / scale + 1, '*');
            end if;
            WhichCol := WhichCol + 1;
        end loop;

        for C in UpperCase loop
            if Uppers(C) /= 0 then
                Plot(WhichCol, 21, Uppers(C) / scale + 1, '*');
            end if;
            WhichCol := WhichCol + 1;
        end loop;
        Screen.MoveCursor(Row => 24, Column => 1);
    end Histogram;
```

## Multidimensional Arrays

So far we have looked only at one dimensional arrays; however, we frequently deal with abstract concept of multidimensional arrays :

- ➢ a chessboard, a Tic_Tac_Toe board, a matrix, Rubik's cube, a map, etc.

In Ada we declare a miltidimensional array in such the same fashion as we do a one dimensional array :

```
type MultArray  is array ( subscript1, . . . , subscriptN) of element_type;
```

Example :

```
        type GameSymbol is (X, O, E);
        type BoardArray is array ( 1 .. 3, 1 .. 3 ) of GameSymbol;

        TicTacToe : BoardArray;

        type YearByMonth is array ( 1900 .. 19999, Month ) of Real;
        type Election is array ( Candidate , Precinct ) of Integer;
        type VideoArray is array ( 1 .. 1024, 1 .. 1024 ) of Pixels;
        type Buffer is array ( 1 .. MaxRow, 1 .. MaxCol ) of Character;
```

## Manipulation of Two - Dimensional Arrays

| Columns | | | | | |
|---|---|---|---|---|---|
| | Index | 1 | 2 | 3 | 4 |
| | 1 | 75 | 62 | 58 | 76 |
| | 2 | 86 | 92 | 90 | 95 |
| Rows | 3 | 78 | 84 | 87 | 89 |
| | 4 | 60 | 72 | 58 | 87 |
| | 5 | 98 | 94 | 93 | 99 |

The Two Dimensional array Scores

To print out all the values in the first row :

```
for Column in 1 .. 4 loop
        Text_IO.Put ( Item => Scores ( 1.. Column ), Width => 4);
end loop;
```

To print out all the values in the first column :

```
for Column in 1 .. 5 loop
        Text_IO.Put ( Item => Scores ( Row .. 1 ), Width => 4);
end loop;
```

## Manipulation of Two - Dimensional Arrays

➢ To sum up all the values in the first row :

```
Sum := 0;
for Column in 1 .. 4 loop
    Sum := Sum + Scores( 1 .. Column);
end loop;
```

➢ To sum up all the values in the first column :

```
Sum := 0;
for Row in 1 .. 5 loop
    Sum := Sum + Scores( Row .. 1);
end loop;
```

➢ To sum up all the values in the array :

```
Sum := 0;
for Row in 1 .. 5 loop
    for Column in 1 .. 4 loop
        Sum := Sum + Scores( Row .. Column);
    end loop;
end loop;
```

**Multidimensional Array Types**

**type** Table **is array** ( 1 .. 5, 1 .. 5 ) **of** Integer;

Jack, Jill : Table;

➢ Indexing in arrays

```
Jack (1, 1) := 10;
Jack (1, 2) := Jack (1, 1);
Index := 3;
Jack ( Index, Index ) := 5;
Jack ( Index -1, 5) := Jack (1, 1) + 5;
Jack (4,3) := 2 * Jack (3, 3) - 3;
```

➢ Array aggregates

```
Jack := ((1,2,3,4,5),
         (2,2,2,2,,2),
         (3,3,3,3,3),
         (4,4,4,4,4),
         (5,5,5,5,5));

Jack := ( 3 => (1,2,3,4,5),
          2 => (2,2,2,2,2),
          1 => (3,3,3,3,3),
          4 => (4,4,4,4,4),
          5 => (5,5,5,5,5));
```

```
Jack := Table' ( 3 => (1,2,3,4,5),
                 2 => (2,2,2,2,2),
                 others => (3,3,3,3,3));

Jack := Table' ( 3 => (1,2,3,4,5),
                 2 => (2,2,2,2,2),
                 others => (3,3, others => 0));

Jack := Table' ( others => ( others => 0));
```

## Abstract Data Types ( ADT's)

### What is an Abstract Data Type ?

- An ADT is just a formal name for what Ada calls a type : a set of values and a set of operations that are appropriately applied to those values.

- A program that uses and ADT is called **a client program.**

- A client program can declare and manipulate objects of the data type and use the data type's operators without knowing the details of the internal representation of the data type or the implementation of the operators ( in other words the details are hidden),

### How are they constructed?

- ADT's are built using Ada packages.

- Constructor - creates an object of the specified type by putting the necessary parts together.

- Selector - Selects a particular component of the object.

- Inquiry - Ask whether the object has a particular property ( e.g. empty ).

- Input/Output - Provides for input and output of the object.

## Private Types

A private type is provided by a package, in order to give clients exactly the operations desired by the designer, and no others. Given a private type, a client can :

- declare variables of that type.
- declare records or arrays that have fields or components of that type.
- use the universal assignment ( := ) to copy a value of one variable to another.
- use the universal assignment ( =, /= ) to test variables for equality and inequality.
- use any other operations provided in the package specification.

A client cannot do anything else with objects of a private type >

- no field selection.
- no arithmetic ( even if the private type is numeric ).
- no array indexing.

Example

```
type <Name> is private;
```

**Abstract Types and Subprograms**

➢ An abstract type is a tagged type (record) intended for use as a parent type for type extensions, but which is not allowed to have objects of its own.

Example

```
package  P  is

        type  T  is abstract tagged null record;

        . . .

end P;
```

➢ An abstract subprogram is a subprogram, which has no body but is intended to be overridden at some point when it is inherited. It can not be called directly or indirectly.

Example

```
package  P  is

        type  T  is abstract tagged null record; -- abstract type

        procedure Op ( X : T ) is abstract; -- abstract subprogram

        or procedure Op ( X : T ) is  <>; -- an other declaration of an

                                          -- abstract subprogram

        . . .

end P;
```

➢ An abstract type is not allowed to have an invisible abstract operation since otherwise it could not be overridden.

( Note :More about abstract types later in Object Oriented programming with Ada.)

## Case Study : Helping Your Cousin with Fractions

### Problem

Your cousin in junior high school is studying fractions and is interested in whether a computer can "do fractions". You wish to show your cousin that a computer can indeed handle rational numbers.

### Analysis

To be useful, the problem solution should provide for creating, reading, and displaying a rational number, for extracting the numerator and denominator parts.

The package should also contain operations for performing rational arithmetic ( addition, subtraction, multiplication, and division ).

It is also useful to provide operations for comparing two rationals : =, /=, <, <=, and >=. This is a problem for which an abstract data type is a good solution, since an ADT provides a type ( in this case Rational ) and a set of operations applicable to that type.

There should also be operations to read and display rational values.

**Case Study : Cont'd**

Design

We will construct an abstract data type package to represent the data structure for a rational number with operators for each of the tasks listed above.

We will represent each rational quantity as a record with numerators and denominator fields, and we will make the rational type **private** so as to prevent client programs from directly manipulating the fields.

We can use Ada's predefined assignment, equality, and inequality for rationals, but this is meaningful only if we store all rationals in lowest terms.

To understand why, remember that Ada's predefined equality compares two records by checking whether each field of one record is equal to the corresponding field of the other. If each comparison yields a true result, the overall equality is true.

If our design did not require rationals to be in lowest terms, then the equality check would return incorrect results; for example, 2/3 = 6/9 is true in the "real world" but would be false in our system. However, if 6/9 were never actually stored in our system, but replaced with its reduced equivalent, 2/3, this problem could not arise.

**Case Study : Cont'd**

```
package Rationals is

-- Specification of the abstract data type for representing
-- and manipulating rational numbers.

        type Rational is private;

        --Operators
        function "/" (X : Integer; Y : Integer) return Rational;
        -- constructor: returns a rational number in lowest terms
        -- Pre : X and Y are defined  -- Post: returns a rational number
        --   If Y > 0, returns Reduce(X,Y)
        --   If Y < 0, returns Reduce(-X,-Y)
        --   If Y = 0, raises ZeroDenominator

         ZeroDenominator: exception;

        function Numer (R : Rational) return Integer;
        function Denom (R : Rational) return Positive;

        -- selectors: return the numerator and denominator of a rational number R
        -- Pre: R is defined
        -- Post: Numer returns the numerator of R; Denom returns the
        -- denominator
```

```
procedure Get (Item : out Rational);
-- Reads a pair of integer values into rational number Item.
-- Pre : none
-- Post: The first integer number read is the numerator of Item;
--       the second integer number is the denominator of Item.
--       "/" is called to produce a rational in reduced form.


procedure Put (Item : IN Rational);
-- Displays rational number Item.
-- Pre : Item is assigned a value.
-- Post: displays the numerator and denominator of Item.


function "+"(R1 : Rational; R2 : Rational) return Rational;
function "-"(R1 : Rational; R2 : Rational) return Rational;
function "*"(R1 : Rational; R2 : Rational) return Rational;
function "/"(R1 : Rational; R2 : Rational) return Rational;
-- constructors: return the rational sum, difference, product,
--   and quotient of rational numbers R1 and B
-- Pre : R1 and R2 are assigned values
-- Post: return the rational sum, difference, product, and
--   quotient of R1 and R2.
```

```
        function "<" (R1 : Rational; R2 : Rational) return Boolean;
        function ">" (R1 : Rational; R2 : Rational) return Boolean;
        function "<="(R1 : Rational; R2 : Rational) return Boolean;
        function ">="(R1 : Rational; R2 : Rational) return Boolean;
        -- inquiry operators: comparison of two rational numbers
        -- Pre : R1 and R2 are assigned values
        -- Post: return R1 < R2, R1 > R2, R1 <= R2, and R1 >= R2, respectively

        private

        -- R1 record of type Rational consists of a pair of integer values
        -- such that the first number represents the numerator of a rational
        -- number and the second number represents the denominator.

        type Rational is
                record
                        Numerator     : Integer := 0;
                        Denominator   : Positive := 1;
                end record; -- Rational
end Rationals;
```

**Case Study : Con'd**

```
with Text_IO;
package body Rationals is
-- Body of the abstract data type for representing
-- and manipulating rational numbers.
-- local function GCD
package My_Int_IO is new Integer_IO ( Num => Integer);

    function GCD(M: Positive; N: Positive) return Positive is
    -- finds the greatest common divisor of M and N
    -- Pre: M and N are defined
    -- Post: returns the GCD of M and N, by Euclid's Algorithm
        R : Natural;
        TempM: Positive;
        TempN: Positive;
    begin -- GCD
        TempM := M;
        TempN := N;
        R := TempM REM TempN;
        while R /= 0 loop
            TempM := TempN;
            TempN := R;
            R := TempM REM TempN;
        end loop;
        return TempN;
    end GCD;
```

```
-- Operators
function "/" (X : Integer; Y : Integer) return Rational is
-- constructor: returns a rational number in lowest terms
-- Pre : X and Y are defined
-- Post: returns a rational number
--   If Y > 0, returns Reduce(X,Y)
--   If Y < 0, returns Reduce(-X,-Y)
--   If Y = 0, raises ZeroDenominator
        G: Positive;

begin

        if Y = 0 then
                raise ZeroDenominator;
        end if;
        if X = 0 then
                return (Numerator => 0, Denominator => 1);
        end if;
        G := GCD(abs X, abs Y);
        if  Y > 0 then
                return (Numerator => X/G, Denominator => Y/G);
        else
                return (Numerator => (-X)/G, Denominator => (-Y)/G);
        end if;

end "/";
```

```
function Numer (R : Rational) return Integer is
begin
        return R.Numerator;
end Numer;

function Denom (R : Rational) return Positive is
begin
        return R.Denominator;
end Denom;

-- selectors: return the numerator and denominator of a rational number R
-- Pre: R is defined
-- Post: Numer returns the numerator of R; Denom
--returns the denominator
procedure Get (Item : out Rational) is
-- Reads a pair of integer values into rational number Item.
-- Pre : none
-- Post: The first integer number read is the numerator of Item;
--      the second integer number is the denominator of Item.
--      "/" is called to produce a rational in reduced form.
        N: Integer;   D: Integer;
begin -- Get
        My_Int_IO.Get(Item => N);
        My_Int_IO.Get(Item => D);
        Item := N/D;
end Get;
```

```
procedure Put (Item : in Rational) is
-- Displays rational number Item.
-- Pre : Item is assigned a value.
-- Post: displays the numerator and denominator of Item.
begin -- Put
        My_Int_IO.Put(Item => Numer(Item), Width => 1);
        Text_IO.Put(Item => '/');
        My_Int_IO.Put(Item => Denom(Item), Width => 1);
end Put;

function "+"(R1 : Rational; R2 : Rational) return Rational is
        N: Integer;
        D: Positive;
begin
        N := Numer(R1) * Denom(R2) + Numer(R2) * Denom(R1);
        D := Denom(R1) * Denom(R2);
        return N/D;  -- compiler will use fraction constructor here!
end "+";

function "*"(R1 : Rational; R2 : Rational) return Rational is
        N: Integer;
        D: Positive;
begin
        N := Numer(R1) * Numer(R2);
        D := Denom(R1) * Denom(R2);
        return N/D;  -- compiler will use fraction constructor here!
end "*";
```

```
function "-"(R1 : Rational; R2 : Rational) return Rational is
begin -- stub
        return 1/1;
end "-";

function "/"(R1 : Rational; R2 : Rational) return Rational is
begin -- stub
        return 1/1;
end "/";

-- constructors: return the rational sum, difference, product,
--   and quotient of rational numbers R1 and B
-- Pre : R1 and R2 are assigned values
-- Post: return the rational sum, difference, product, and
--   quotient of R1 and R2.

function "<" (R1 : Rational; R2 : Rational) return Boolean is
begin
        return Numer(R1) * Denom(R2) < Numer(R2) * Denom(R1);
end "<";

function ">" (R1 : Rational; R2 : Rational) return Boolean is
begin -- stub
        return True;
end ">";
```

```
        function "<=" (R1 : Rational; R2 : Rational) return Boolean is
        begin -- stub
                return True;
        end "<=";

        function ">=" (R1 : Rational; R2 : Rational) return Boolean is
        begin -- stub
                return True;
        end ">=";

        -- inquiry operators: comparison of two rational numbers
        -- Pre : R1 and R2 are assigned values
        -- Post: return R1 < R2, R1 > R2, R1 <= R2, and R1 >= R2, respectively

end  Rationals;
```

**Case Study : Con'd**

```
with Text_IO;
with Rationals;

procedure TestRational1 is
-- Tests the package Rationals
        A: Rationals.Rational;
        B: Rationals.Rational;
        C: Rationals.Rational;
        D: Rationals.Rational;
        E: Rationals.Rational;
        F: Rationals.Rational;
begin -- TestRational1

        A := Rationals."/"(1,3);
        B := Rationals."/"(2, -4);
        Text_IO.Put(Item => "A = ");
        Rationals.Put(Item => A);
        Text_IO.New_Line;  Text_IO.Put(Item => "B = ");
        Rationals.Put(Item => B);
        Text_IO.New_Line;

        -- Read in rational numbers C and D.
        Text_IO.Put(Item => "Enter rational number C as 2 integers > ");
        Rationals.Get(Item => C);
        Text_IO.Put(Item => "Enter rational number D as 2 integers > ");
        Rationals.Get(Item => D);
        Text_IO.New_Line;
        E := Rationals."+"(A,B);              -- form the sum
        Text_IO.Put(Item => "E = A + B is ");
        Rationals.Put(Item => E);
        Text_IO.New_Line;

        F := Rationals."*"(C,D);              -- form the product
        Text_IO.Put(Item => "F = C * D is ");
        Rationals.Put(Item => F);
        Text_IO.New_Line;
        Text_IO.Put(Item => "A + E * F is ");
        Rationals.Put(Item => Rationals."+"(A, Rationals."*"(E,F)));
        Text_IO.New_Line;
end TestRational1;
```

## Variant Records - General Form

- All field names must be unique.
- An empty field list is indicated by a null instead of a field list.
- All values of the discriminant must be covered by **when** clauses. A discriminant is used to state which variant of record is being dealt with

```
type Face ( Bald : Boolean ) is
    record
        Eyes   : Color ;             -- start of fixed fields
        Height : Inches;
        case Bald is                 -- start of field lists
            when True =>
                WearWig : Boolean;
            when False =>
                HairColor : Color;
        end case;
    end record;
```

➢ A derived type only can have discriminants.

➢ Examples

1. The parent type does not  have discriminants and the new type has.

```
type Person is tagged
        record
                Name : String ( 1.. 30 );
                NameLength : Natural := 0;
        end record;

type Employee ( StartingSalary : Natural ) is new Person with
        record
                Salary : Natural := StartingSalary;
        end record;
```

When an employee is declared a StartingSalary must be stated

```
E : Employee(1800);
```

2. The parent type has discriminants but the derived non

```
type Temp_Employee is new Employee with
        record
                Start, Finish : Time;
        end record;
```

The derived type inherits the discriminant of its parent. So when we declare a temporary employee we have to give a value for the discriminant StartingSalary.

```
TE : Temp_Employee (1600);
```

3. Both the parent and the derived type have discriminants.

```
type Permanent_Employee ( First_Salary : Positive;
                No : Positive) is new Employee ( First_Salary) with
        record
                date_of_Employment : Time;
        end record;
```

The permanent_Employee has no discriminant called StartingSalary and so it does not inherit the component Salary. It gets the discriminants that are given in the new declaration First_Salary.

The discriminant StartingSalary for the Employee always has the same value as the discriminant First_Salary in the derived type

Permanent_Employee. When we declare a variable of type Permanent_Employee we must give values to the two discriminants First_Sarary and No

```
PE : Permanent_Employee ( 1600, 1234);
```

➢ The discriminant of a variant part is allowed to be a generic formal type

➢ A value for a discriminant field must be supplied unless a default ( := ) is provided in the definition.

```
type Face ( Bald : Boolean := True ) is
record
    Eyes   : Color ;            -- start of fixed fields
    Height : Inches;
    case Bald is                -- start of field lists
        when True =>
            WearWig : Boolean;
        when False =>
            HairColor : Color;
    end case;
end record;
```

## Constrained / Unconstrained Variant Records

- An unconstrained record variable is one that has a default discriminant value, and none is supplied in the variable declaration.

```
JohnsFace : Face;
```

- A constrained record variable is one whose discriminant value is supplied when the variable is declared

```
JohnsFace : Face( Bald => Flse;
```

- It is not permitted to change the discriminant value of a constrained record at execution time.

```
JohnsFace: Face ( Bald => False )     -- declaration

            -- later in the program

JohnsFace := ( Bald => True)          -- Cannot do this
```

## Storing Values into Variant Records

- Any field may be selected and read at any time.

- Any field may be selected and changed except a discriminant field if the change is not consistent with the discriminant value.

- The discriminant field of a constrained record cannot be changed under any circumstances.

- The discriminant field of an unconstrained record can be changed only if the entire record is changed at the same time ( aggregate or copy ).

**In otherwords - It is best to supply a default value for the discriminant because unconstrained variant records are more flexible!**

## Operation on Variant Records

- A variable record value may always be assigned to an unconstrained variable of the same record type ( it is permissible to change the discriminant of an unconstrained variable ).

- A variant record value can be assigned to a constrained variable of the same type only if the discriminant values match (the discriminant value of a constarined variable can never be changed ).

- Two variant record values can be compared for equality only if the discriminant values agrre.

```
type KidKind is (Girl, Boy);

type Child ( Sex : KidKind := Girl )  is
    record
        First : Character;            -- start of fixed fields
        Last : Character;
        Age  : Natural;
        case  Sex  is
            when Girl =>
                Sugar  : Float;
                Spice   : Float;
            when  Boy =>
                Snakes  : Integer;
                Snails   : Integer;
                Tails     : Integer
        end case;
    end record;
```

## Unconstrained Array Types

### What is an unconstrained array?

- An unconstrained array is one declared in such a way that the bounds of the array are not specified in the type declaration.

  ```
  type ListType  is array ( Integer range <>) of  Float;
  ```

  In this case ( Integer range <>) means the subscript range ( bounds) must be a subrange of the integers.

- The bounds are supplied only when variables of that type are created.

  ```
  L1  : ListType ( 1 .. 50 );
  L2  : ListType ( -10 .. 10 );
  L3  : ListType ( 0 .. 20 );
  ```

**Operations on Unconstrained Array Types**

➢ The operations of assignment and equality testing are defined : however,

(1) both operands must be a variable of the same unconstrained array type.

(2) both must have the same number of elements.

```
type ListType  is array ( Integer range <>) of  Float;
L1  : ListType ( 1 .. 50 );
L2  : ListType ( -10 .. 10 );
L3  : ListType ( 0 .. 20 );

L2 := L3;        -- OK same size for all onconstrained array types

L1( 20 .. 40 ) := L2;   -- OK same size and type

L2( 1 .. 5 ) := L1( 6 .. 10 );    -- same size and type

L1 := L2;        -- will raise constraint error ( different sizes )
```

**Attribute Functions**

```
type ListType  is array ( Integer range <>) of  Float;
L1  : ListType ( 1 .. 50 );
L2  : ListType ( -10 .. 10 );
L3  : ListType ( 0 .. 20 );
```

L2'First returns the lower bound of L2 or -10 in this case

L2'Last returns the upper bound of L2 or 10

L2'Length returns the number of elements in L2 or 21

L2'Range returns the range -10 .. 10

These attribute functions are frequently quite useful :

```
for  i  in L2'Range  loop
    My_Flt_IO.Put ( Item => L2(i), Fore => 1, Aft => 2, Exp => 0);
    Text_IO.New_Line;
end loop;
```

**Examples of Unconstrained Array Usage**

```
with Text_IO;

procedure TestMaxValue is

    type  ListType is array(Integer range <>) of Float;
    L1 : ListType(1..5);     -- 5 elements
    L2 : ListType(-4..3);    -- 8 elements
    package My_Flt_IO is new Float_IO ( Num => Float);

    -- local procedure to display the contents of a list
    procedure DisplayList(L: ListType) is
    -- display the contents of a list, represented as an unconstrained array
    -- Pre: L is defined
    -- Post: display all values in the list

    begin -- DisplayList

        for Count in L'Range loop
            My_Flt_IO.Put(Item=>L(Count), Fore=>3,
                                                Aft=>1, Exp=>0);
        end loop;
        Text_IO.New_Line;
    end DisplayList;
```

```
function MaxValue(L: ListType) return Float is
-- return the largest value in an object of type ListType
-- Pre: L is defined
-- Post: returns the largest value stored in L
        CurrentMax : Float;
begin -- MaxValue


        CurrentMax := Float'First;    -- minimum value of Float
        for WhichElement in L'Range loop
                if L(WhichElement) > CurrentMax then
                        CurrentMax := L(WhichElement);
                end if;
        end loop;
        -- assert: CurrentMax contains the largest value in L
return CurrentMax;
end MaxValue;


begin -- TestMaxValue
        L1 := (0.0, -5.7, 2.3, 5.9, 1.6);
        L2 := (3.1, -2.4, 0.0, -5.7, 8.0, 2.3, 5.9, 1.6);

        Text_IO.Put(Item=> "Testing MaxValue for float lists");
        Text_IO.New_Line;
        Text_IO.New_Line;
        Text_IO.Put(Item=> "Here is the list L1");

        Text_IO.New_Line;  DisplayList(L => L1);
```

```
          Text_IO.Put(Item=> "The maximum value in this list is ");
          My_Flt_IO.Put(Item => MaxValue(L=>L1), Fore=>1,Aft=>2,Exp=>0);
          Text_IO.New_Line;
          Text_IO.New_Line;
          Text_IO.Put(Item=> "Here is the list L2");
          Text_IO.New_Line;  DisplayList(L => L2);
          Text_IO.Put(Item=> "The maximum value in this list is ");
          My_Flt_IO.Put(Item => MaxValue(L=>L2),   Fore=>1, Aft=>2, Exp=>0);
          Text_IO.New_Line;
      end TestMaxValue;
```

## Examples of Unconstrained Array Usage

```
package Vectors is

        -- specification for vector arithmetic package
        type Vector is array (Integer range <>) of Integer;

        -- exported exception, raised if two vectors are not conformable
        -- (i.e., have different bounds)
        Bounds_Error : exception;

        function "+" (K : Integer; Right : Vector) return Vector;
        -- adding a scalar to a vector
        -- Pre: K and Right are defined
        -- Post: returns the sum of the vector and the scalar
        --   Result(i) := K + Right(i)

        function "*" (K : Integer; Right : Vector) return Vector;
        -- multiplying a vector by a scalar
        -- Pre: K and Right are defined
        -- Post: returns the product of the vector and the scalar
        --   Result(i) := K * Right(i)

        function "*" (Left, Right : Vector) return Integer;
        -- finds the "inner" or "dot" product of two vectors
        -- Pre: Left and Right are defined and have the same bounds
        -- Post: returns the inner product of Left and Right
```

```
    function "+" (Left, Right : Vector) return Vector;
    -- finds the sum of two vectors
    -- Pre: Left and Right are defined and have the same bounds
    -- Post: returns the sum of Left and Right
    --    result(i) := Left(i) + Right(i)
end  Vectors;
```

## Examples of Unconstrained Array Usage

```
package Matrices is
        -- specification for package Matrices
        type Matrix is array(Integer range <>, Integer range <>) of Float;
        -- exported exception, raised if two matrices are not conformable

        Bounds_Error : exception;

        function "+" (K : in Float; M : in Matrix) return Matrix;
        -- adds a scalar to a matrix
        -- Pre: K and M are defined
        -- Post: returns the sum of the scalar and the matrix
        --   Result(i,j) := K + M(i,j)

        function "*" (K : in Float; M : in Matrix) return Matrix;
        -- multiplies a matrix by a scalar
        -- Pre: K and M are defined
        -- Post: returns the product of the scalar and the matrix
        --   Result(i,j) := K * M(i,j)

        function "+" (Left, Right : in Matrix) return Matrix;
        -- finds the sum of two matrices
        -- Pre: Left and Right are defined and have the same bounds
        -- Post: returns the sum of Left and Right
        --   Result(i,j) := Left(i,j) + Right(i,j)
        --   Raises Bounds_Error if the matrices are not conformable
```

```
    function "*" (Left, Right : in Matrix) return Matrix;
    -- finds the product of two matrices
    -- Pre: Left and Right are defined
    --   and Left's column bounds agree with Right's row bounds
    -- Post: returns the product of Left and Right
    --   Raises Bounds_Error if the matrices are not conformable

    function Transpose(M : in Matrix) return Matrix;
    -- finds the transpose of a matrix
    -- Pre: M is defined
    -- Post: returns a matrix such that Result(i,j) = M(j,i)
    --   Result has M's bounds, interchanged

end Matrices;
```

## Generic Units

- A generic component ( package or subprogram ) is one that is parameterized at the level of the types that it works with. There are generic formal and actual parameters, just like the ones we use with subprograms. In other wotds, a generic component can be instantiated ( tailored ) to work with variety of different types.

- Let's look at the familiar procedure below designed to exchange the values of two variables of type Natural :

```
procedure  Exchange ( Value1, Value2 : in out Natural) is
        TempValue : Natural;
begin
        TempValue := Value1;
        Valu1 := Value2;
        Value2 := TempValue;
end Exchange;
```

- The same sequence of statements are used to accomplish an exchange of values of two variables of type float:

```
procedure  Exchange ( Value1, Value2 : in out Float) is
        TempValue : Natural;
begin
        TempValue := Value1;
        Valu1 := Value2;
        Value2 := TempValue;
end Exchange;
```

## Generic Units

- To exchange any the values of a given type the sequence of statements are always going to be the same; that is they constitute a recipe or template for exchanging the values of any given type.

- Ada's generic facility allow us to take advantage of the generalized nature of routines and allow us to create a single routine to handle many different types.

```
generic
          type ValueType is private;    -- formal type
          procedure GenericSwap (Value1, Value2: in out ValueType);


          procedure  GenericSwap ( Value1, Value2 : in out ValueType) is
          TempValue : Natural;
begin
          TempValue := Value1;
          Valu1 := Value2;
          Value2 := TempValue;
end Exchange;
```

- Now the generic is ready to be instantiated ( tailored ) by plugging in the type ):

```
generic
          type ValueType is private;
          procedure GenericSwap (Value1, Value2 : in out ValueType );
          procedure IntegerSwap is new GenericSwap (ValueType => Integer);
          procedure CharSwap is new GenericSwap (ValueType => Character);
```

**Generic Units**

➢ The formal type in the generic unit can be one of the following :

1. **type** Item **is private**;
2. **type** Buffer ( Length : Natural ) **is limited private**;
3. **type** Enum **is** (<>);
4. **type** Int **is range** <>;
5. **type** Angle **is delta** <>;
6. **type** Mass **is digits** <>;
7. **type** Table **is array** (Enum) **of** Item;
8. **type** Point **is access** Node;
9. **type** Employee **is tagged private**;
10. **type** Employee **is tagged limited private;**
11. **type** Permanent_Employee **is new** Employee **with private**;
12. **type** Tree **is abstract null record**;

➢ Generic subprogram parameter ( or formal subprogram) :

1. **with** subprogram_declaration;

   with Compare ( Value1, Value2 : ValueType) return Boolean;

2. **with** subprogram_declaration **is** subprogram_name;

   **with procedure** Upadate **is** Default_Update;

3. **with** subprogram_declaration **is** <> ;

   **with function** "<" ( E1, E2 :Element) **return** Boolean **is** <> ;

4. **with** package parametter_name **is new** Gen_Pack ( <> );

   Gen_Pack is another generic package.

```
generic
        with package Parameter is new P1 ( <> );
package P2 is
. . .
end P2;
```

## Generic Subprogram Parameters

- Sometimes a generic needs to be instantiated with the names of functions or procedures or other generic packages. For instance, lte's look at our old friend Maximum which returned the larger of its two integer operands.

```
function Maximum ( Value1, Value2 : Integer ) return Integer is
        Result : Integer;
begin
        if Value1 > Value2 then
                Result := Value1;
        else
                Result := Value2;
        end if;
        return Result;
end Maximum;
```

- Ideally we would like to make a function that returns the larger of its two operands regardless of the types of the operands.

- The additional problem we encounter here that we did not encounter with our GenericSwap is the addition of the comparison operation :

```
        if Value1 > Value2 then
```

- Suppose the type which we are comparing does not have obvious predefined greater-than operator. For instance suppose we are comparing personnel records in an array of records ; how do you compare them? By name? By Zip Code ? etc.

**Generic Subprogram Parameters**

```
function GenericMaximum ( Value1, Value2 : ValueType ) return ValueType is
          Result : Integer;
     begin
          if Compare(Value1, Value2) then
               Result := Value1;
          else
               Result := Value2;
          end if;
          return Result;
     end GenericMaximum;


generic
  type ValueType is private;
  with Compare(Valu1, Value2 : ValueType ) return Boolean;
 function GenericMaximum ( Value1, Value2 : ValueType ) return ValueType is
```

➢ We can instantiate ( or tailor ) or generic for differing types :

```
     function   Maximum  is new
          GenericMaximum( ValueType => Integer, Compare => ">");
     function   Maximum  is new
          GenericMaximum( ValueType => Float, Compare => ">");
     function   Minimum  is new
          GenericMaximum( ValueType => Integer, Compare => "<");
     function   Maximum  is new
          GenericMaximum( ValueType => Float, Compare => "<");
```

**Generic Subprogram Parameters**

- But note that the operators "<" and ">" are predefined for the types that we have instantiated ( that is the Integer and Float types come pre-equipped with the less-than and greater-than operators).

- What happens if we need to compare two variables for which these operations are not predefined? <u>We must define them</u>!

- Let's take a quick look at a generic sorting routine.

**Generic Sorting Procedure**

```
-- procedure specification for GenericSwapSort

generic -- procedure specification for GenericSwapSort
        -- here are all the generic formal parameters

        type ElementType is private;  -- any nonlimited type will do
        type IndexType   is (<>);     -- any discrete type for index
        type ListType    is array (IndexType RANGE <>) of ElementType;
        with function Compare (Left, Right : ElementType) return Boolean;


        procedure GenericSwapSort(List: in out ListType);

                -- procedure body for GenericSwapSort
                with GenericSwap;             -- context clause
                procedure GenericSwapSort(List: in out ListType) is

                -- we need to make an instance of GenericSwap for this case
                procedure Exchange is new GenericSwap (ValueType =>
                                                              ElementType);
```

```
begin -- GenericSwapSort
    for PositionToFill in List'First..List'Last loop

    -- Store in List(PositionToFill) the "largest" element remaining
    -- in the subarray List(PositionToFill..List'Last)
        for ItemToCompare in PositionToFill..List'Last loop
            if Compare(List(ItemToCompare),
                        List(PositionToFill)) then
                Exchange(List(PositionToFill),
                            List(ItemToCompare));
            end if;
        end loop;
    -- assert: element at List(PositionToFill) is "largest" in subarray
    end loop;
end GenericSwapSort;
```

## Generic Sorting Procedure

➢ If we had a program that contained the following declarations, we might desire to sort an array of student score records :

```
MaxSize : constant Positive := 250;
MaxScore : constant Positive := 100;
subtype StudentName is String(1..20);
subtype ClassIndex is Positive range 1..MaxSize;
subtype ClassRange is Natural range 1..MaxSize;
subtype ScoreRange is Natural range 1..MaxSize;

type ScoreRecord is
        record
                Name : StudentName;
                Score  : ScoreRange;
        end record;

function ScoreLess ( Score1, Score2 : ScoreRecord) return Boolean is
begin
        return Score1.Score < Score2.Score;
end ScoreLess;

procedure SortScores is new GenericSwapSort
                        ( ElementType => ScoreRecord,
                        IndexType => ClassIndex,
                        ListType => ScoreArray
                        Compare => ScoreLess);


          Scores : ScoreArray ( ClassIndexFirst .. ClassIndexLast);
          ClassSize : ClassRange;
```

➢ Now we can make the call to do the sorting :

```
          SortUpScores ( List => Scores(1 .. ClassSize);
```

**Recursion**

- The general concept of a subprogram ( a procedure or function ) calling itself is known as recursion.

- Although recursion often provides us with a rather elegant solution to a problem, it comes with a price ( time and space ) resulting from the overhead associated with additional procedure calls.

- Recall, that when a procedure or function is called a new activation record for that subprogram is pushed onto the program stack. The current value of all the local variables as well as the address of the next line of the code to be executed is recorded prior to actually placing the record on the stack.

- Despite the additional overhead incurred, recursion often provides a simple, natural and elegant solution to many types of problems. These solutions are generally much easier to read and understand than iterative solutions to the same problem, a benefit which can justify the additional overhead costs.

**Recursive Algorithms**

➢ Recursive algorithms are generally of the form :

```
if (the stopping case is raised ) then
        solve the problem
else
        reduce the problem using recursion
end if;
```

➢ The following simple example for performing multiplication illustrates the general nature of recursive programming.

➢ The function Multiply, which returns the product of M x N, splits the original problem into two simpler problems :

1. Multiply M by N - 1
2. Add M to the result
3. The stopping case is when N is finally reduced to 1.

**Recursive Multiplication Function**

```
function Multiply ( M : in Integer, N : in Positive ) return Integer is

-- Performs multiplication recursively using the + operator
-- Pre : M and N are defined and N > 0
-- Post : returns M * N

        Result : Integer;
begin  -- Multiply

        if  N = 1  then
                Result := M;    -- stopping case
        else
                Result := M + Multiply ( M, N-1 );    --recursion
        end if;

        return  Result;

end Multiply;
```

**Test Program**

```
with Text_IO;

procedure TestMultiply is

        FirstInt  : Integer;
        SecondInt : Positive;
        Answer    : Integer;
        package My_Int_IO is new Integer_IO ( Num => Integer);

        function Multiply (M : in Integer; N : in Positive) return Integer is
        -- Performs multiplication recursively using the + operator
        -- Pre : M and N are defined and N > 0
        -- Post: returns M * N
                Result: Integer;
        begin  -- Multiply
                Text_IO.Put(Item => "Multiply called with parameters");
                My_Int_IO.Put(Item => M);
                My_Int_IO.Put(Item => N);
                Text_IO.New_Line;
                if N = 1 then
                        Result := M;                    -- stopping case
                else
                        Result := M + Multiply(M, N-1);  -- recursion
                end if;
```

```
            Text_IO.Put(Item => "Returning from Multiply with result");
            My_Int_IO.Put(Item => Result);    Text_IO.New_Line;
            return Result;
        end  Multiply;

begin -- TestMultiply

        Text_IO.Put(Item => "Please enter a integer > ");
        My_Int_IO.Get(Item => FirstInt);
        Text_IO.Put(Item => "Please enter a positive integer > ");
        My_Int_IO.Get(Item => SecondInt);
        Answer := Multiply(M => FirstInt, N => SecondInt);
        Text_IO.Put(Item => "The product of the two integers is ");
        My_Int_IO.Put(Item => Answer, Width => 1);
        Text_IO.New_Line;

end TestMultiply;
```

## Tracing a Recursive Function

- Lte's trace the execution of our Multiply test program by examining the values recorded on the activation records for each recursive call of the function. Real activation records do not look exactly like our drawings but these drawings serve to illustrate how we can trace recursive subprograms.

```
function Multiply ( M : in Integer, N : in Positive ) return Integer is
Result : Integer;
begin  -- Multiply
        if N = 1 then
                Result := M;   -- stopping case
        else
                Result := M + Multiply ( M, N-1 );   --recursion
        end if;
        return Result;
end Multiply;
```

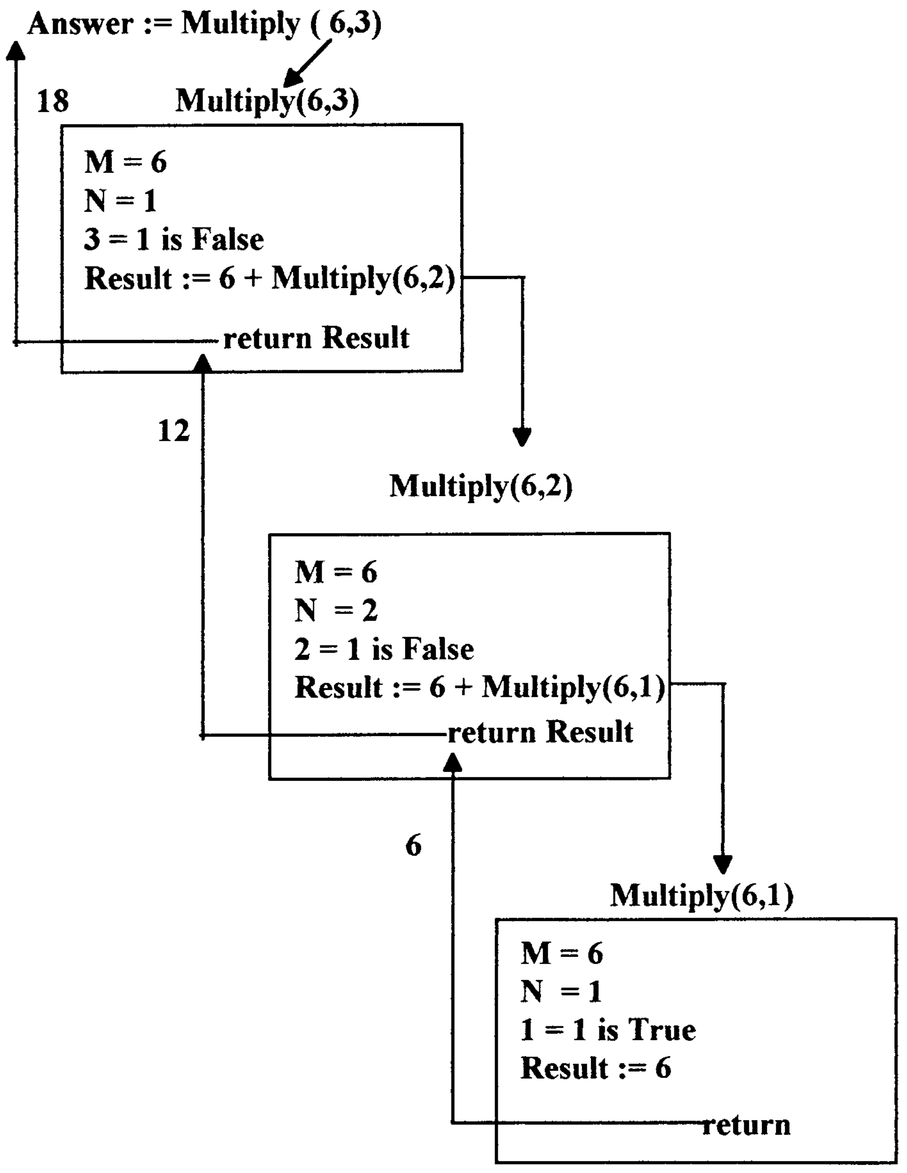
Client program call -> Answer := Multiply ( 6, 3 )

**Multiply (6,3)**

```
M = 6
N = 1
3 = 1 is False
Result:= 6 + Multiply (6,2)
        Return Result
```

## Tracing a Recursive Function

➢ Recall that with each function call an activation record is filled in then "pushed" onto the program stack and that with each return an activation record is "popped" from the stack.



**Another Recursion Example**

- The two Foactorial functions below illustrate both an iteratively designed solution to the problem of finding factorials and one designed to solve the problem recursively.

**Iterative Solution**

```
function  Factorial ( N : in Natural ) return Positive is
          Fact : Natural := 1;
begin
          if  N = 0  then
                    return  1;
          else
                    for i  in  1 .. N  loop
                              Fact := Fact * i;
                    end loop;
                    return Fact;
          end if;
end Factorial;
```

Recursive Solution

```
          function  Factorial ( N : in Natural ) return Positive is
          begin
                    if  N = 0  then
                              return  1;        --stopping case
                    else
                              return  N * Factorial ( N - 1);-- split using recursion
                    end if;
          end Factorial;
```

**Trace of Factorial**(3)

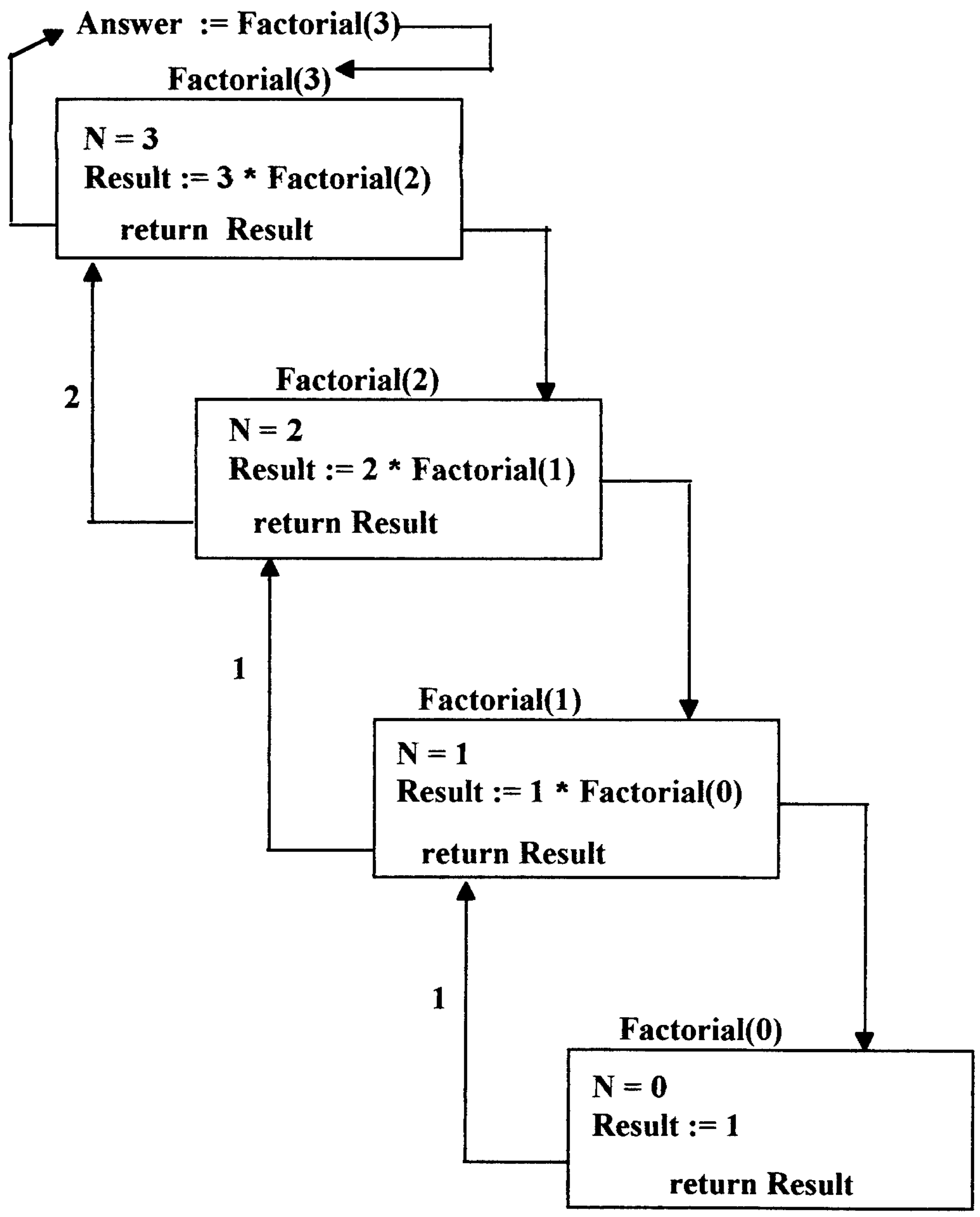
```
function  Factorial ( N : in Natural ) return Positive is
     begin
          if  N = 0  then
               return  1;       --stopping case
          else
               return  N * Factorial ( N - 1);-- split using recursion
          end if;
     end Factorial;
```

Client program call  -->  Answer := Factorial (3);

**Factorial (3)**

```
+------------------------------+
|    N = 3                     |
|    Result := 3 * Factorial(2)|
|                              |
|        return Result         |
+------------------------------+
```

Trace of Factorial(3)



## A Final Note on Recursion

- Most errors in recursive programs are infinite loops which are caused by missing or incorrect stopping conditions. Always make certain you check to see that the stopping condition is valid.

- For large problems with many recursive calls, it is not uncommon to run out of stack space. You may have to increase the size of your program stack ( compiler / environment options ) prior to running your program.

- You may also find it worthwhile to look at QuickSort ( O ( N log(N)) ) algorithm which is much faster than our old friend the SelectionSort ( O ( $N^2$ ) )

## Dynamic Data Structures

- A dynamic data structure is a collection of elements (nodes) that are generally implemented ae records or subprograms. Space for these records is allocated at run time (from a pool of free memory called the heap). The individual nodes are then connected (linked) through the use of memory address pointers.

- These address pointers are referred to in Ada as access variables. Variables of this type simply hold the memory address of a for referencing variables created dynamically off heap.

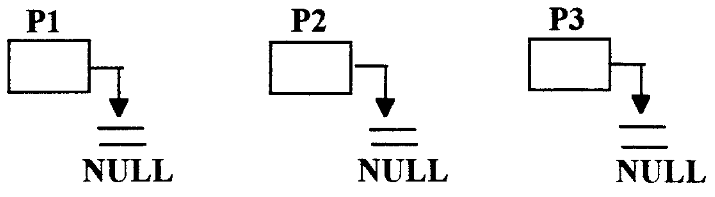
- Dynamic data structures can grow or shrink during program execution.

- Consider a record type as follows :

```
type  RecType  is
        record
                . . .      -- fields
        end record;
```

- The type definition below gives us the ability to declare variables of type RecPointer, that is, variables that can hold pointers to things of type RecType.

```
type RecPointer is access RecType ;
P1, P2, P3 : RecPointer;        -- create three variables
```



**P1, P2, P3 initialized to NULL**

➢ There are two kinds of access type :

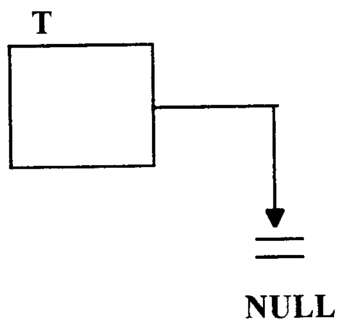
1. access to object types ( the example in the previous page) and
2. access to subprograms.

➢ Access to subprograms

Example.

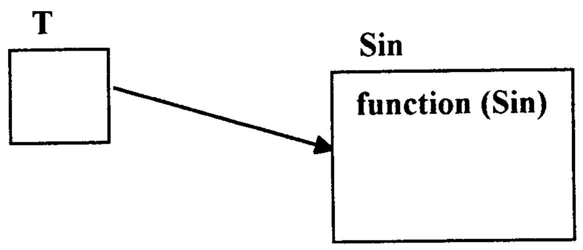
type Trig_Function is access function(R : Real ) return Real;

T : Trig_Function;      --create a variable



➢ T point to a function Sin as loow:

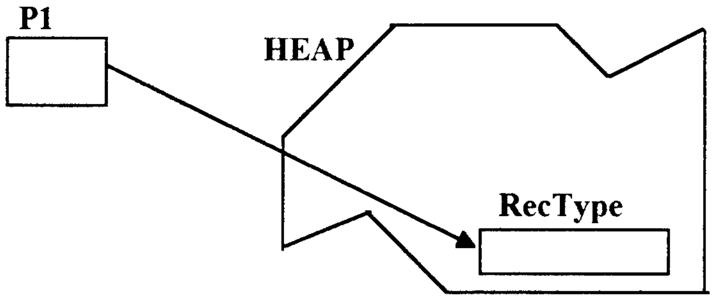
**function** Sin( R : Real ) **return** Real;
T : Sin'Access;



- When an access variable is created in Ada its value is **always** initialized to a special unique internal value known as NULL. This indicates that the pointer does not point to anything.

- The new operator is used to dynamically create new variables. The following statement dynamically allocates space from the heap to create a new RecType variable and stores the address (reference) to that variable in P1. P1 is now said to point to the new variable.

```
P1 : new  RecType;
```



- Access variables can be copied using the assignment operator. The statement :

```
P3 := P1;
```

  causes P3 to point to the same address as P1.

- Each time we make the call P1 := NewType, a new address is pcaced in P1. This means that if we are not careful we can easily lose track of dynamically allocated variables. These variables are then unaccessible, both for our use and for use by the heap manager.

- More about access types in the Object Oriented Programming with Ada.

## Dynamic Data Structures

Creating a Linked Structure

- Before we go any ferther, the following story may help you to visualize the concept of dynamically linked structures. As you read the story, try to imagine how you would model the story in Ada.

- The following story is completly fictional, any resemblence to actual characters is purely coincidental.

**The Story of Julio's Parking Lot**

Players :

| | |
|---|---|
| Maria | -- very bright, ambitious and extremely attractive |
| Frank | -- Maria's husband, hard working, naive and somewhat of a dullard. |
| Julio | -- Frank's best friend, very wealthy, suave and something of a playboy. |

Background :

Maria, who has worked evenings at the local pub as a cocktail waitress for the past five years, has grown tired of the drudgery of her job and decides that there must be a better way to make a dollar. Being the intelligent and ambitious woman she is, she attempts to convince her husband Frank, that they should take their life savings and invest it in a business of their own.

Frank, who has been working as a sardine packer is not enthusiastic about giving up his job at the sardine packing factory. He argues that his present job, despite the fact that it doesn't pay very well, is both interesting and offers long term job security.

Maria persists and suggests to Frank that they purchase old wrecks and start their own low priced car rental service called "Rent a Junk". Furthermore, she points out to Frank that they can make a real killing since the initial investment would be very low, insurance costs would be minimal and maintenance costs negligible since they could just sell the junk cars back to the salvage yard rather than repair them.

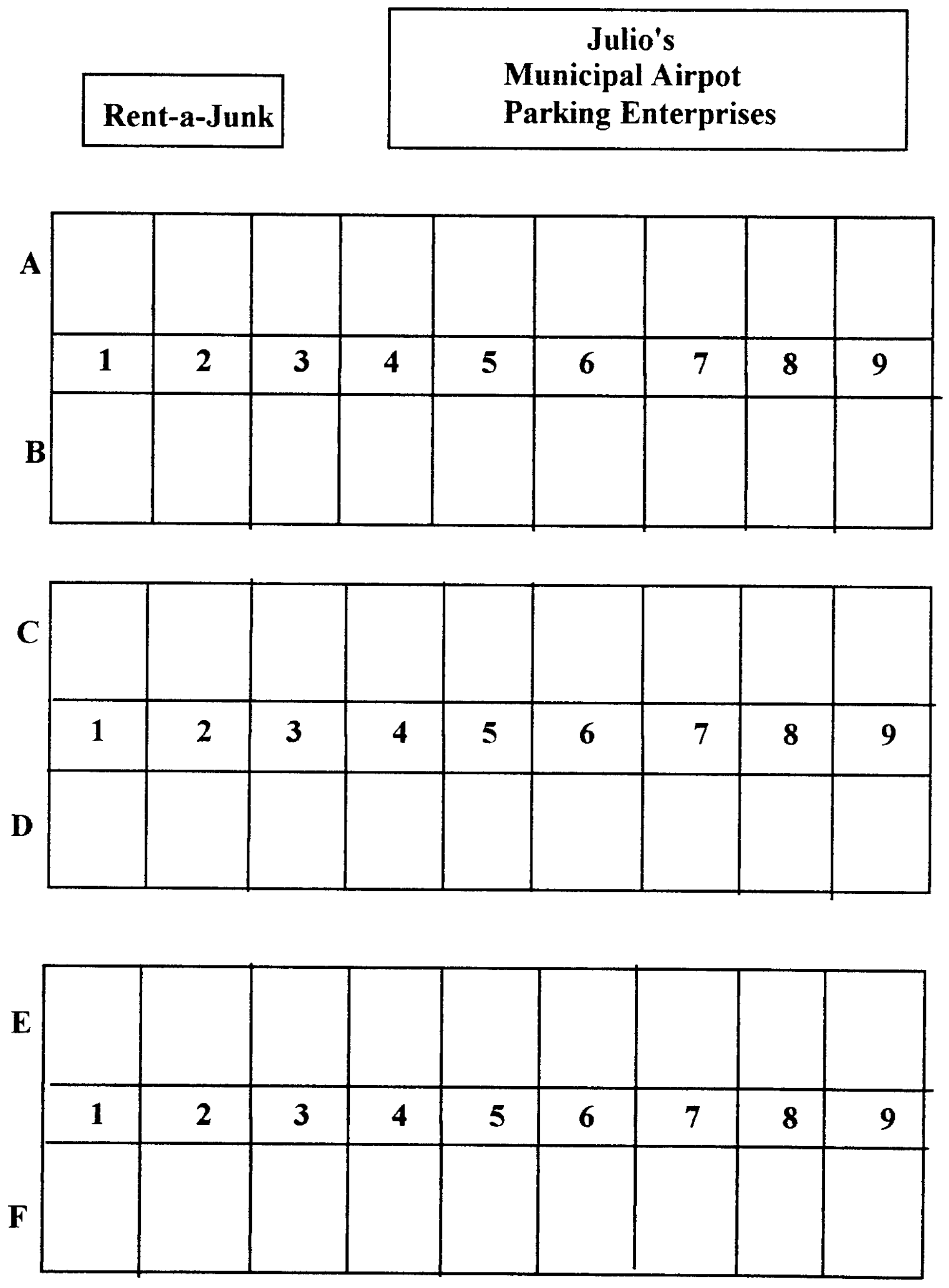
Frank, despite the fact he has never been nominated for the Nobel Prize, accurately criticizes Maria's plan by pointing out that this would require them to have a parking lot to park all the wrecks. He argues that initial start up costs would also include the purchase of a plot of land and a small building from which they could conduct their business. This initial cost, he argues, is prohibitive and suggests that it would be smarter for them both to keep their current jobs.

Maria of course has already found a solution to the problem. Maria tells Frank that she has worked out an arrangement with Frank's best friend Julio the owner of the huge parking lot at the municipal airport. For a minor fee of $1.00 per day per car, they can park their cars in available parking spaces, to be specified by Julio. Furthermore, she says, Julio has agreed to let them use an old runway line shack to run their business from, for a very small fee ( and an occasional amorous favor ). The line shack is located only a few feet away from Julio's swank new office, which will make it easy for Maria to contact Julio to obtain the location of available parking spaces and to negotiate payment.

Julio, who enjoys Maria's company ( and favors ), insists that parking spaces must be negotiated for one at a time, since must check his computer to verify that any vacant spots have not been contacted out by his staff. (Devilish grin)

Frank reluctantly agrees to give up his job and to start the new business, only after Maria exhausts all efforts at reasoning and resorts to more effective and amorous negotiations on her dull wilted husband. Several months later, Frank is balancing himself precariously on the roof of the extremely small converted runway line shack attempting to secure the new "Rent a Junk" sign while Maria is making final arrangements for delivery of their first shipment of junkers.

**Rent - Junk -(Two Weeks Before Opening)**



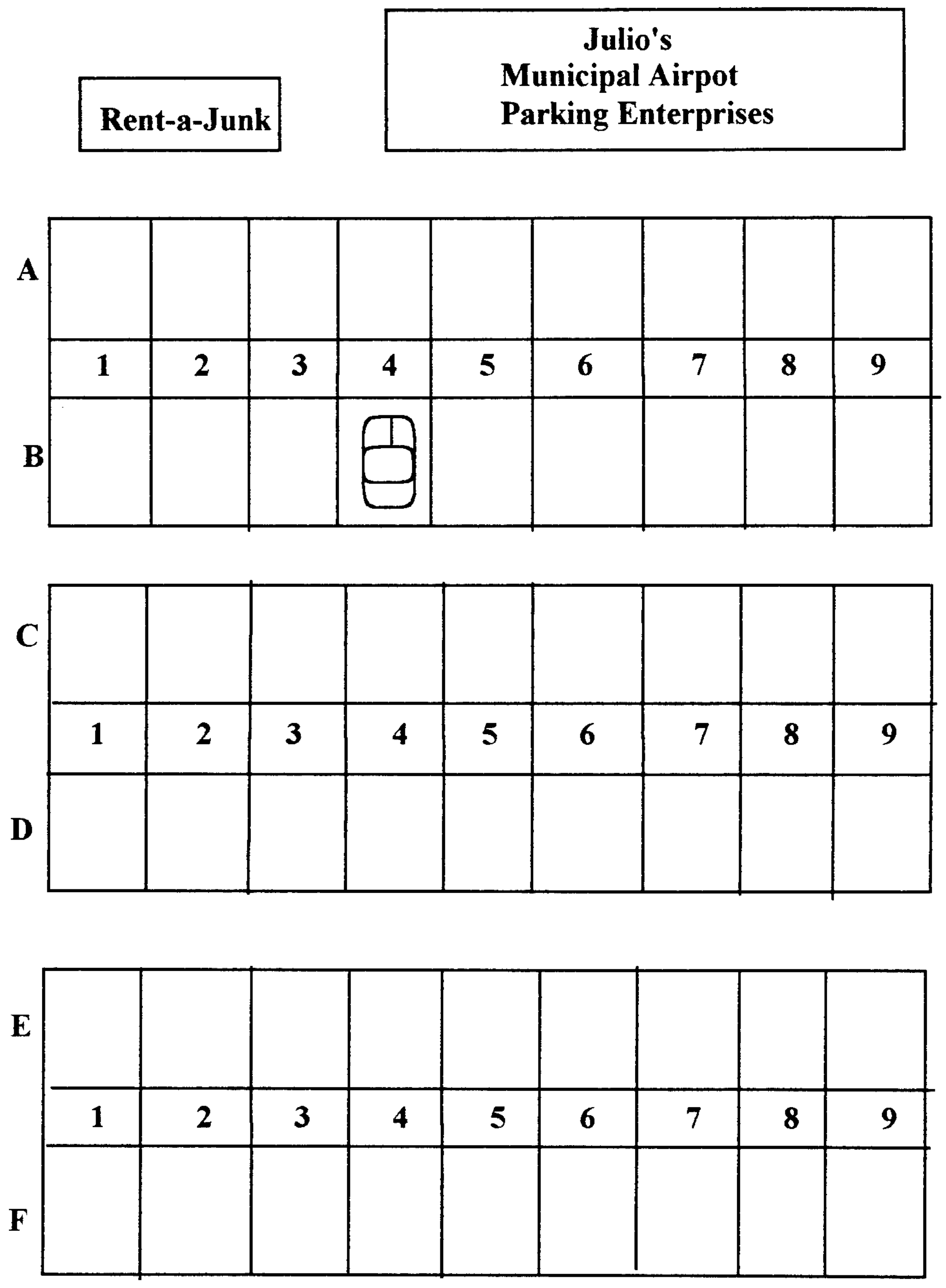
## Single Linked Lists

Several days later the first shipment of junk cars arrive and Maria tells Frank she is going next door to get a parking space. Maria enters the swank office and calls out in an enticing but subdued voice: "Oh Julio"

Maria returns after a few brief, but passionate, moments with number of an unassigned parking slot to place the first junker. She calls out Frank with number of the parking space. Unfortunately, poor Frank being the dullard that he is, can only remember three things at any given time and since his poor memory is already overtaxed just ensuring that he can remember his own as well as his beautiful wife's name, he asks Maria to write to write the location down on a piece of paper.

Maria knowing that is almost certain that Frank will lose the piece of paper, writes the location on the back of his right hand and sends Frank out to park the first car.

**Rent - Junk -(Two Weeks Before Opening)**



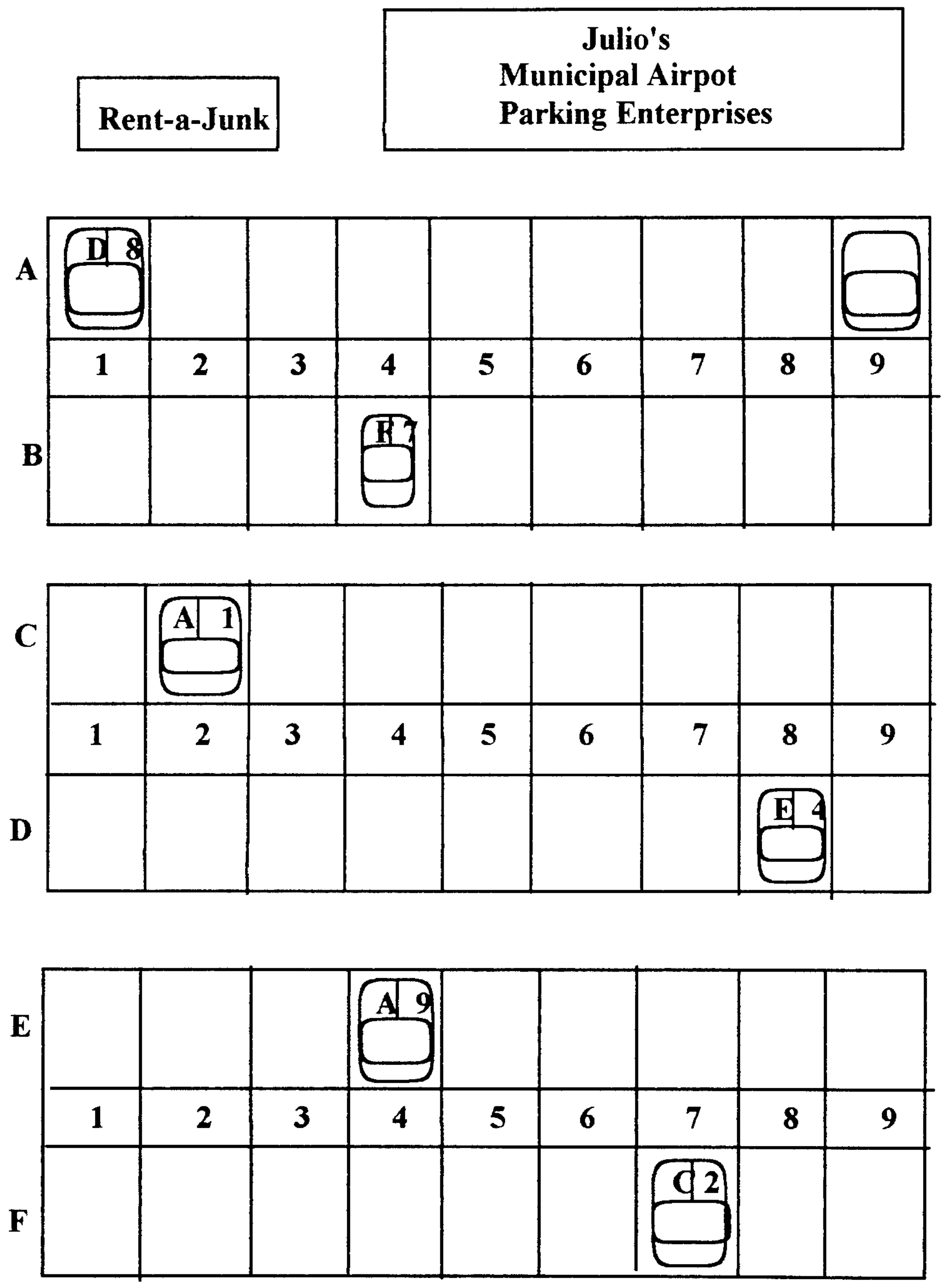
**Rent - a - Junk**

Frank gleefully returns and announces to Maria that he has done as she has directed in reply to which. Maria tells Frank to unload the second car from the trailer while she goes next door to obtain another parking space. "Oh Julio" she cries!... then returns.

Again she tells Frank to go and park the new junker, realizing Frank's propensity for forgetfulness, she instructs him to return to the car parked in the space written on his right hand. Then directs him to write the number of the new parking space on the hood of the car parked in the location. She explains to Frank that in the future be will only have to recall where the first car is parked in order to find all the others that he has parked, because all he has to do is go to the first car, read the location of the second car off the hood of the first, go to that location and read the location of the third car off the hood, and continue in this fashion until he reaches the last car which will not have a number on the hood.

Frank, dizzied by his wife's brilliance and beauty, steps out once again to do she has directed. The same routine continues until the last of the cars have been removed from the trailer and parked in their spaces.

**Rent - a - Junk ( First Shipment Parked )**



**Rent - a- Junk**

Maria then hands poor old Frank a bucket of soapy water, a sponge, a mobile fresh water tank, a DustBuster and directs Frank to go wash and vacuum the cars before they officially open for business the next morning. Maria sweetly encourages her poor husband to finish quickly by promising an interesting evening of amorous favors in appreciation for all of his hard work. Frank, happily proceeds on his way to the first car.

## Frank's Dilemma #1

After Frank has washed all the cars, he realizes that he has must have lost his wallet inside one of the cars while he was cleaning the interior. How will he find his wallet?

**Answer**

Start at the first car and follow the numbers written on the hoods of the car until he finds the wallet he is searching for or he arrives at the end of the list of cars ( recall that he will know when he has reached the last car because there will be nothing written on the hood of the car).

## Frank's Dilemma #2

The next day a customer comes in to "Rent a Wreak" to rent any old junker that they can provide. What directions will Maria give to Frank in order to allow him to receive a car for their first customer?

**Answer**

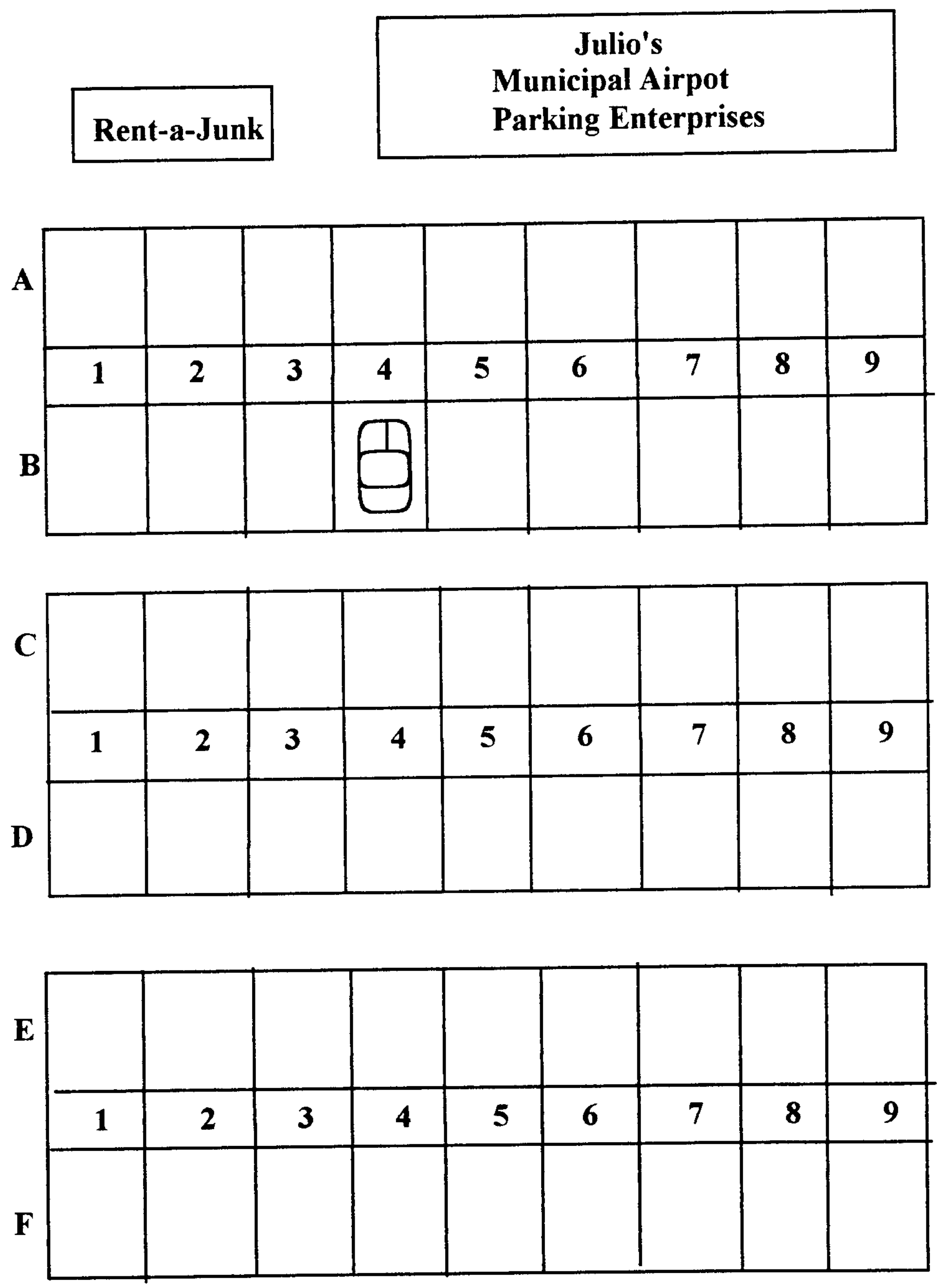
Return with the car located in the parking space written on the back of his right hand. Before giving the car to the customer, erase the number written on the back of his hand and replace it with the number that is written on the hood of the car.

## Doubly Linked Lists

Unimpressed by the story Frank relates to her about searching for his wallet. Maria devises a new schema for parking the cars that should make it easier for dear old Frank to find his wallet in the future. Since all the cars have been rented out, she decides that the best time to implement her schema is upon the return of the first vehicle.

She now gives Frank the parking space number and tells Frank to park the car and, once again, write down the number she has just given him on the back of his right hand.

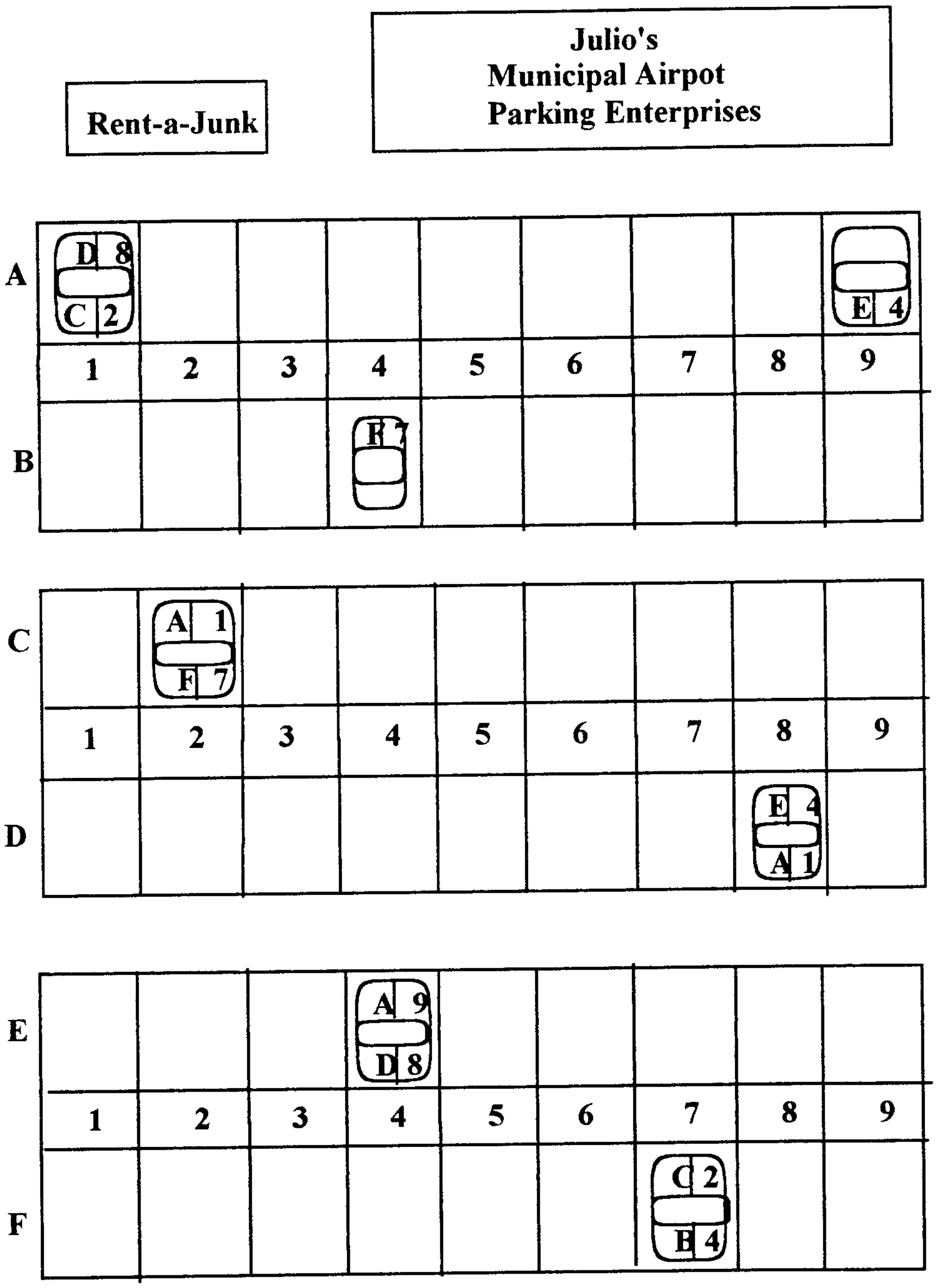
Rent - a - Junk ( First Rental Returns )



**Rent - a - Junk**

When the second car returns Maria instructs Frank to return to the car parked in the space written on his right hand, then directs him to write the number of the new parking space on the hood of the car parked in that location. However, this time she also tells him to write the number of the parking space of the first car on the trunk of the car he is currently parking before proceeding to the new parking spot.

Rent - a - Junk (All Rental Returned)



**Rent - a - Junk**

Again she explains to Frank that in the future be will only have to recall where the first car is parked in order to find all the others since all he has to do is go to the first car, read the location of the second car off the hood of the first, travel to that location and read the location of the third car off the car in that spot and so forth just as he did before. In addition, should be again lose his wallet he will not have to return to the first car in the list and search all the subsequent cars. She further explains that be merely has to glance at the number written on the trunk of the last car and start going backwards in the list of cars until he finds his wallet or reaches the first car in the list which he will recognize as it will not have a number written on the trunk.

**Epilog**

"Rent a Junk" was a successful that our dashing young Julio was overcome with exhaustion from all of Maria's favors and passed away, but with such a smile of entertainment on his face that all who attended his funeral noticed with amazement. Before he died however, Julio was so pleased with their business arrangement that he willed his entire parking lot enterprise to his good friend Frank.

Frank finally discovered the true nature of Maria's business arrangement with Julio after which they discovered, sold their business to another equally enterprising and energetic young couple, Mario and Francine.

Today, Maria is the proud owner of the local pub in which she previously worked and Frank now runs his old friend Julio's airport parking lot business!

"OH, FRANK" . . .

(Frank turns in Francine's direction and grins from ear to ear!!)

# APPENDIX B

## OBJECT ORIENTED PROGRAMMING DEFINITION

Object Oriented programming is a way of thinking about process of decomposition problems and developing programming solutions.

Object Oriented Programming builds upon the following :

➢ Object Oriented design which contains :

- ► Objects, Entities that have structure and state.
- ► Operations which are actions on objects that may access or manipulate the state.
- ► Encapsulation that is some means of defining objects and their operations and providing an abstract interface to them, while hiding their implementation details.

➢ Inheritance. It is a mean for incrementally building new abstractions from existing one by "inheriting" their properties without disturbing the implementation of the original abstraction or the existing clients.

➢ Polymorphism( From the Greek poly, many, and morphe, form). It is a mean of factoring out the differences among a collection of abstractions such that programs may be written in terms of their common properties. It has the ability to identify a type at run time and to manipulate values of several specific types.

## ADA 94 GENERAL APPROACH TO OOP

Ada 83 supports an object oriented design but does not have the support for inheritance and polymorphism found in fully object oriented languages.

Ada 94 contains all the object oriented design features from Ada 83 and in addition inheritance and polymorphism facilities.

The **inheritance** is provided by the type extension features that are expressed through the tagged types and child library units.

There are two means, of using polymorphism in Ada 94 .

➢ **Static polymorphism** that is provided through the generic parameter mechanism whereby a generic unit may at compile time be instantiated with any type from a class of type.

➢ **Dynamic polymorphism.** It is provide through the use of so-called class-wide types. Also the dynamic polymorphism introduces the ;

► **Late binding** . It is the ability to choose an operation at run time ( this choice is made late in the compile link run process. The late binding is provided by the following :

◆ **Dispatching.** The choice of subprogram to call is made at run time depending on the type of the 4 parameters or possibly the type of the result of the subprogram call.

◆ **Access to Subprogram.** We have access type which can reference subprograms.

## PROGRAMMING BY EXTENSION

Programming by extension provides the ability to declare a new type that refines an existing parent type by inheriting, modifying or adding to both the existing components and the operations of the parent type.

Programming by extension is provided by the tagged types.

Tagged types are record or private types. So a tagged record or private type maybe extended with additional components on derivation.

➢ Record tagged types

```
type Object is tagged
    record
        X_Coord : Real;
        Y_Coord:  Real;
    end record;
```

Type Object is a tagged record that can be extended to other types ( e.g. Circle type). We can declare an extended type Circle in two ways with:

► Additional Components

```
type Circle is new Object with
    record
        Radius : Real;
    end record;
```

Type Circle is derived from the Object type and has one additional component (Radius) and also inherits all the operations related with this object.

► No Additional Components

```
type Circle is new Object with null record;
```

The declared type Circle in this case is derived from the object type but does not have any additional components.

➢ Private type

We can extend a private type Object as

```
type Object  is  tagged private;

. . .

private

    type Object is tagged

    record

        X_Coord : Real;

        Y_Coord:  Real;

    end record;
```

We can declare a type Circle the same as the tagged type records;

► With Additional Components

```
type Circle is new Object with

record

    Radius : Real;

end record;
```

► With no Additional Components

```
type Circle is new Object with null record;
```

The tagged types( e.g. Object type) can be declared in package specification or in the declarations on the package body.

The type extension type ( e.g. Circle type) is not allowed at place which is not accessible from the tagged type ( or parent type ) for example within an inner block.

We can declare two tagged types in the same package but we cannot in that package declare a subprogram that has operands or result of both type. We allow to do it only outside the package.

The operations closely related to a tagged type to be inherited must be in the visible part of the package. The following example shows such a situation.

```
package  Geometry is
    type Object is tagged
    record
        X_Coord : Real;
        Y_Coord:  Real;
    end record;

    function Distance ( O : in Object) return Real is
        begin
            return Sqrt ( O.X_Coord ** 2 + O.Y_Coord ** 2);
        end Distance;

    type Circle is new Object with
    record
        Radius : Real;
    end record;

end Geometry;
```

In the above example the function Distance is inherited because is in the visible part of the package. The type Circle is derived from the type Object with one additional component Radius and also inherits the function Distance.

The following  example shows an illegal extension of a tagged type.

```
package P is
        type T is tagged . . .
        procedure OP1( X : T);
end P;

with P;
use P;
procedure K is
        package Inner is
                type NT is new T with -- Illegal extension because the tagged type( or
                                                parent type T is not visible from the Inner package
                        record
                                . . .
                        end record;
                procedure OP1( X: NT);
        end Inner;
        package body Inner is
                . . .
                procedure OP1( X: NT) is
                        begin
                        . . .
                        end OP1;
                . . .
        end Inner;
begin
. . .
end K;
```

The following example shows a legal extension for a record type

```
package P is
type T is tagged . . .
      procedure OP1( X : T);
end P;


with P;
use P;
package Q is
      type NT is new T with -- legal extension of type T. Type is visible from this
                              -- point .
            record
            . . .
            end record;
      procedure OP1( X : NT);
end Q;
```

**Record Aggregates**

Record aggregates are only permitted for a type extension if both the extension part and the parent part are fully visible. In other words we can only use an aggregate where we can view all the components.

```
type Object is tagged
        record
            X_Coord : Real;
            Y_Coord:  Real;
        end record;


type Circle is new Object with
        record
            Radius : Real;
        end record;
```

We can make the following aggregations for the types Object and Circle of course by taking care the visibility rules that we mentioned earlier.

```
O : Object := (1.0, 0.5);
C : Circle  := ( 0.0, 0.0, 34.7);
O: Object( C );
C: Circle := ( O with 41.2 ); or C: Circle := ( Object with Radius => 41.2);
```

## CLASS WIDE PROGRAMMING

Each tagged type has an associated type denoted by T'Class.

T'Class is declared implicitly whenever a tagged record type is defined.

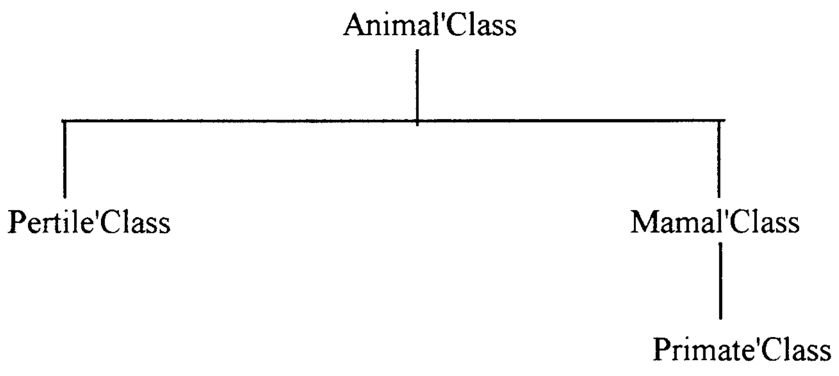
T'Class comprises the union of all the types in the tree of derived types rooted at T.

The values of T'Class are the values of T and all its derived types.

The type T'Class is treated as an unconstrained type; this is because we cannot possibly know how much space could be required by any value of a class wide type because the type might be extended.

```
type Animal is tagged
    record
        Species : Species_Name;
        Weight : Grams;
    end record;
type Mammal is new Animal with
    record
        Hair_Color : Color_Enum;
    end record;
type Reptile is new Animal with
    record
        Poisson_On_Mouth : Boolean;
    end record;
type Primate is new Mammal with
    record
        Female_Born_Property : Boolean;
    end record;
```

The tree structure from the above types is :



The Animal is a parent type for Reptile and Mammal and Mammal is a parent type for the Primate.

The Mammal is not a subtype of Animal; Mammal and Animal are distinct types and values of one type cannot be directly assigned to objects of the other type.

Class wide types have no operations of their own. However, users may define explicit operations on class-wide types. An example of such an operation is a procedure Print.

**procedure** Print( A : **in** Animal'Class);

Procedure Print may be applied to any object within the class of animals.

We can declare an access type referring to a class wide type. The access type designates any value of the class wide type from time to time.

**type** Animal_Ptr **is access** Animal'Class.

The benefit from the use of the class-wide types is a programmer can define several operations having the same name, even though each operation has a different implementation.

**Class-Wide Operations**

```
type Animal is tagged
    record
        Species : Species_Name;
        Weight : Grams;
    end record;
type Mammal is new Animal with
    record
        Hair_Color : Color_Enum;
    end record;
A : Animal'Class;
```

We have the following operations :

```
if A in Mammal then
        . . . -- special processing for Mammal
    end if;
```

When A is of the class Mammal execute the body of if statement.

```
If A in Mamal'Class then
        . . . -- special processing for Mammal
    end if;
```

We can also test the tag explicitly using the attribute Tag. So we could write.

```
if A'Tag = Mammal'Tag then
    . . . -- special processing for Mammal
end if;
```

## ABSTRACT TYPES AND SUBPROGRAMS

Abstract type is a tagged type intended for use as a parent type for type extensions, but which is not allowed to have objects of its own.

```
type Set is abstract tagged null record;
```

The purpose of the abstract type is to provide a common foundation upon which useful types can be built by derivation.

Upon derivation from an abstract type we can provide actual subprograms of the parent type.

Abstract subprogram is a sort of place holder for an operation to be provided ( it does not have body). Function Union is an example.

**function** Union ( Left, Right : Set) **return** set **is abstract;** or

**function** Union ( Left, Right : Set) **return** set **is** <>**;** Symbol <> is same as the abstraction reserved word in Ada 94.

**Example**

```
package Base_Reservation_System is
    type Reservation is tagged null record;
    procedure Make( R : in out Reservation ) is <> ;
end Base_Reservation_System;
with Base_Reservation_System;
package Subsonic_Reservation_System is
    type Position is (Aisle, Window);
    type Meal_Type is (Green, White, Red);
    type Basic_Reservation is new Base_Reservation_System.Reservation with
        record
            Flight_Number  : Integer;
            Date_Of_Travel: Date;
            Seat_Number    : String(1..4):="    ";
        end record;
    procedure Make( BR : in out Basic_Reservation);
    procedure Select_Seat( BR : in out Basic_Reservation);
    type Nice_Reservation is new Basic_Reservation with
        record
            Seat_Sort  : Position;
            Food       : Meal_Type;
        end record;
    procedure Order_Meal ( NR : in Nice_Reservation);
    procedure Make( NR : in out Nice_Reservation );
    type Posh_Reservation is new Nice_Reservation with
        record
            Destination : Address;
        end record;
    procedure Arrange_Limo(PR : in Posh_Reservation);
    procedure Make (PR : in out Posh_Reservation);
end Subsonic_Reservation_System;
```

In this example procedure Make is an abstract subprogram so we can enforce it for all derived types Basic_Reservation, Nice_Reservation and Posh_Reservation.

When we derive from an abstract type we do not have to provide a proper subprogram for every abstract one. However, if we do not, then the newly derived type will also be abstract.

If we derive from a non-abstract type we can provide abstract operations ( either additional ones or to replace inherited ones ) and as a consequence the derived type will then be abstract.

## DISCRIMINANTS

A discriminant is a parameter of a composite type. It can control for example the bounds of a component of the type if that type is an array type. A discriminant of a task type can be used to pass data to a task of the type upon derivation.

```
type Gender is (Male, Female);
type Person (Sex, Gender) is tagged
    record
        Birth : Date;
    end record;
```

and we can extend Person to type Man

```
type Man is new Person (Male) with
    record
        Bearded : Boolean;
    end record;
```

The type Man inherits the discriminant from Person in the sense that a man still has a component called Sex although of course it is constrained to be Male.

We can have two discriminants in one type. In the following example type Boxer provides a new set of discriminants.

```
type Weight is (Light, Middle, Heavy);
type Boxer (W: Weight) is new Person (Male) with
    record
        . . .  -- information according to weight
    end record;
```

We can also declare :

```
package D is
      type T is private;
      . . .
private
      type T ( N: Natural := 0) is
      . . .
end D;
```

This declaration help us to solve the problem of ragged arrays and varying strings by declaring a private type with out a discriminant and make the full type discriminated, so that we can choose access type storage for very long strings and direct storage for short strings.

Also we can declare a private type with unknown discriminants

```
      type T (<>) is limited private;
```

The user can not declare object outside the defining package thereby giving the package complete control over the creation of objects. The user could of course be given access values referring to such objects as the result of calling subprograms in the package.

## OPERATIONS OF TAGGED TYPES

### ➢DISPATCHING

The runtime choice of a procedure is called dispatching. In the following example procedure Print is a dispatching one.

```
procedure Process_Animal ( AC : in out Animal'Class) is
          . . .
begin
          Print( AC ) ; -- dispatch according to tag
          . . .
end Process_Animal;
```

The procedure Print (AC) is a dispatching call because the value of the tag of AC is used to determine which procedure Print to call and this is determined at runtime.

In the following example procedure Print is not a dispatching one because the value of the tag is determined at the call time.

```
procedure Process_Animal( AC : in out Reptile) is
          . . .
begin
          Print ( Animal ( Reptile)); -- is not a dispatching call
          . . .
end Process_Animal;
```

The following example shows when we have dispatching or not. Package Example declares the tagged type T and the dispatching operations of T the procedures P, Q and the functions F, G. Also declares an extension type TT .

```
package Example is
    type T is tagged . . .
    procedure P ( X : T; Y : T);
    function F return T;
    function G( Z : T) return T;
    procedure Q( U : T; V : T := F);
    type TT is new T with . . .;
    . . .
end Example;
```

P, Q, F and G are dispatching operations of T because all controlling operands and results of a call must be of the same type T.

Suppose we have the variables, whose types are as follows:

```
A, B    : T;
AA, BB : TT;
C       : T'Class := . . . ;
D       : T'Class :=. . . ;
```

then

P (A, B); This is non-dispatching , type T because the P is determined by the call .

P( AA, BB ); non-dispatching, type TT.

P( C, D ); Dispatching because we do not know the procedure we use which is determined at runtime . There are the following cases for the procedure P.

1. P( Type_T, Type_T);
2. P( Type_TT, Type_TT);
3. P( Type_Of_Class_T, Type_Of_Class_T);
4. P( Type_Of_Class_TT, Type_Of_Class_TT);

P( A, BB); Illegal because mixes specific types.

P( A, C); Illegal because mixes static( A of T) and dynamic types ( C of T'Class).

P( A, F); Non-dispatching, type T because A is static and determines that the call of F is also static. The call of F is chosen at compile time to be F with the result of type T.

P( C, F); Dispatching because C is dynamic and determines the type at runtime. In this case the call of F dispatches to the particular F with the same type as C.

P( C, G(D)); Dispatching. The tags of C and D are checked to ensure that they are the same; Constrained_Error is raised if they are not;

P( A, G(F)); Non-dispatching

P(C, G(F)); Dispatching. The call of G is then determined by the specific type of C and this in turn determines the call of F.

Q(A); Non-dispatching. The default call of F is statically determined to be that of type T.

Q(C); Dispatching. The call of F is dynamically determined by the specific type of the value C.

C: T'Class := F; non-dispatching. F is statically determined to be T.

P( F,F) Illegal because of inheritance we do not know whether we are dealing with the P and F of the type T or TT. In this case the overload resolution fails. Dis patching is not involved because there are no clacc-wide operands.

## ➢ REDISPATCHING

When after one dispatching operation we apply a further common ( and inherited ) operation and so need to dispatch once more to an operation of the original type this is called redispatching.

```
procedure Print ( MA : in out Mammal) is
    begin
        Print ( Animal(MA));  -- print a mammal
        Manipulate( Mammal'Class(MA)); -- redispatch because first we have to
                                       --know the Mammal type and after to
                                       --Manipulate it.
    end Print;
```

## TYPE CONVERSION

1. Conversion towards the root of a tree.

```
MA : Mammal'Class;
Make_Animal ( Animal(MA);
```

We converted a Mammal into an Animal.

2. Extension in the same type.

```
type Masic_Animal is new Animal with null record;
A : Animal:= . . .;
BA:=(A with null record);
```

3. Conversion of a value of a class-wide type to a specific type.

```
AC : Animal'Class;
MA:= Mammal(AC);
```

The AC is of the type Animal'Class. There is a runtime check that the current value of the class wide parameter AC is of a specific type for which the conversion is possible.

4. View Conversions

The underlying object is not changed but we merely get a different view of it e.g.

```
Make_Animal ( Animal(MA);
```

The value passed on to the call of Make_Animal is in fact the same value as held in MA but we can no longer see the components relating to the type Mammal.

5. Full Conversion

```
PM : Primate'Class;
MA := Mammal(PM);
```

The tag of MA is of course not changed. The components appropriate to the type of MA are copied from the object PM. This is not a view conversion but a full blooded value conversion.

## ACCESS TYPES

A mechanism for late binding we have already seen in the previous lectures is the dispatching. Another mechanism is provided by the manipulation of subprogram values through an extension of access types.

In Ada 94 an access type can refer to a subprogram; such an access to subprogram value can be created by the Access attribute and a subprogram can be called indirectly by dereferencing such an access value.

The access to subprogram is used to program general dynamic selection and to pass subprograms as parameters.

**type** Trig_Function **is access function**( R: Real) **return** Real;

T         : Trig_Function;

X, Theta : Real;

T can point to functions such as Sin, Cos and Tan.

We assign an appropriate access to subprogram value to T by writing:

```
    T:= Sin'Access;
```

We can indirectly call the subprogram currently referred to by T as:

```
    X:= T( Theta ); ( or X:= T.all(Theta); )
```

Access type as a parameter to a function

**type** Integrand **is access function** ( X: Real) **return** Real;

**function** Integrate ( F : Integrand; A, B : Real) **return** Real;

```
Area := Integrate( Log'Access, 1.0, 2.0); -- It computes the area under the curve for
                                          --log(x) from 1.0 to 2.0.
```

Within the body of function Integrate there will be calls of the actual subprogram passed as parameter.

The subprogram that is accessed can have no parameters.

```
type Action is access procedure;
Action_Sequence : array ( 1..N ) of Action;
. . . -- build the array
      -- and then obey it
for I in Action_Sequence'Range loop
      Action_Sequence ( I ).all; -- we have to use .all because there are no
                                 -- parameters.
end loop;
```

➢ **Operation of Access Types**

**X'Access ;** Yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object, including possibly the current instance of a limited type within its definition or a formal parameter of a tagged type.

**P'Access;** Yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (S). The subprogram P shall be accessible from S.

➢ **General Access types**

1. We can read and update a variable.

```
type Int_Ptr is access all Integer;
IP : Int_Ptr; -- we assign a variable IP of type Int_Ptr.
I: aliased Integer;
IP := I'Access;
```

We can read and update the variable I through the access variable IP.

2. We want to have only read access to a variable.

We can use **constant** instead of **all** when we want to restrict the access to be read-only.

```
type Const_Int_Ptr is access constant Integer;
CIP : Const_Int_Ptr;
I   : aliased Integer;
C   : aliased Constant Integer := 1815;

CIP := I'Access; ( or CIP := C'Access;)
```

3.The type accessed by a general access type can be any type such as an array or record. So we can build chains from records statically declared.

```
AI : array ( 1.. 100) of aliased Integer;
. . .
IP := AI(I)'Access;
```

4. The accessed value could be a component of any composite type. Thus we could point into the middle of a record ( provided the component is marked as aliased).

```
type Ref_Count is access constant Integer range 0 .. 1;
type Ref_Count_Array is array ( Integer range <>) of Ref_Count;
type Cells is
      record
            Life_Count : aliased Integer range 0 .. 1;
            Total_Neighbour_Count : Integer range 0 .. 8;
            Neighbour_Count: Ref_Count_Array ( 1..8);
            . . .
      end record;
```

We can link the cells together

```
This_Cell.Neighbour_Count(1) := Cell_To_North.Life_Count'Access.
```

The type Ref_Count and the component Life_Count have the same static subtypes so that they can be checked against each other at compile time.

We can not apply the Access attribute to a component of an unconstrained variable if the component depends upon discriminants.

5. Access discriminants and parameter.

We can have an access value as a discriminant and also as a parameter of a subprogram.

```
procedure Select_Animal ( AN : access Animal );
procedure Print ( A : in out Animal ) is
    begin
        select_Animal( Animal'Class(A)'Access);
    end Print;
```

6. A discriminant of an access type.

It is useful for effectively parameterizing one record with another.

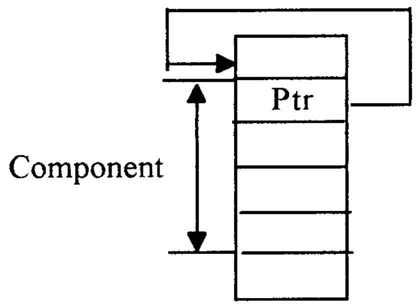
```
type Outer is limited private;
    private
        type Inner ( Ptr : access Outer) is ..
        type Outer is limited
            record
                . . .
                Component : Inner( Outer'Access);
                . . .
            end record;
```

The component of type Inner has an access discriminant Ptr which refers back to the instance of the record Outer.

If we declare

Obj : Outer;

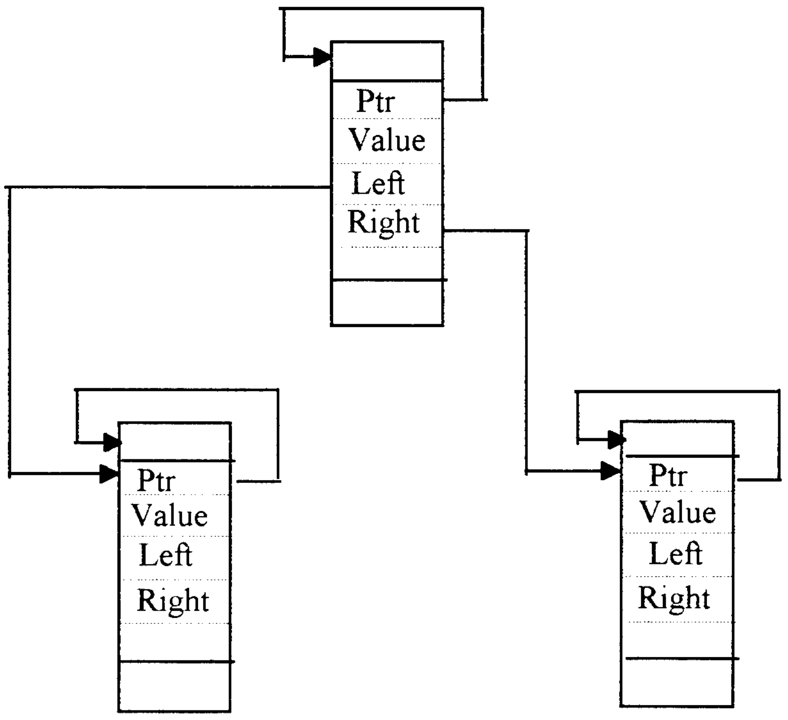
then we create the following structure.



Inner can be extension of other type ( e.g. Node).

type Inner(Ptr : access Outer'Class) is new Node with ...

Then we can navigate over the tree by using the access discriminant Ptr. The structure that is created is :



## HIERARCHICAL LIBRARY

The problems that Ada 83 had with the library package construction were:

- ➢ The coarse control of visibility of private types. When we wish to write two logically distinct packages which nevertheless share a private type.
- ➢ The inability to extend without recompilation. When we wish to extend an existing system by adding more facilities to it.

Ada 94 solves the above problems by the introduction of a hierarchical libraries.

A hierarchical library structure contains child packages and child subprograms.

The introduction of the hierarchical library provides decomposition of a library unit with possibly distinct views of the hierarchy for the client and implementor.

The overall program structure is enhanced by the concept of partitions.

More control of elaboration order is provided.

A library package is only permitted to have a body if one is required by language rules.

The mechanism of the program library is no longer specified so precisely. The main issue is that partitions must be consistent.

**Child Units.**

An example of child library unit :

```
package Complex_Numbers is
        type Complex is private;
        . . .
        function "+" ( X, Y: Complex) return Complex;
        . . .
        function Cons (R, I: Real) return Complex;
        function Ri_Part( X : Complex) return Real;
        function Im_Part( X : Complex) return Real;
private
        . . .
end Complex_Numbers;
```

We can add a child package as follows

```
package Complex_Numbers.Polar is
      procedure Cons_Polar ( R, Theta: Real) return Complex;
      function "abs" (X : Complex) return Real;
      function  Arg (X : Complex) return Real;
end Complex_Numbers.Polar;
```

Child library units have the following general properties.

► Are logically dependent on their parent and have visibility of their parent's visible and private parts. Within the body of Complex_Numbers.Polar we can access the full details of the private type Complex.

► They are named like nested units; with an expanded name consisting of a unique identifier and their parent's name as prefix. Complex_Numbers.Polar has as prefix Complex_Numbers the parent's name and as a unique identifier the Polar.

► A child unit may be any kind of library unit including a generic unit or instantiation.

► This structure may be iterated child units that are packages ( or generic packages ) may themselves have children, yielding a tree like hierarchical structure, beginning at a root library unit. Note however that a generic unit may only have generic children. For example we can have a child of the Complex_Numbers.Polar package, named Complex_Numbers.Polar.A .

Child library units observe the following visibility rules.

► A parent unit's visible definitions are visible everywhere in any child unit, whether the unit is public or private.

► A parent unit's visible definitions are visible in the private part of a child unit, whether the child is public or private.

► A parent unit's visible definitions are visible everywhere in any child unit, since the child package is never visible outside of the parent.

► The entities in a parent's body are never visible in a child unit.

**Private children**

With private children we decompose a system for implementation reasons but without giving any additional visibility to clients.

```
package OS is
    . . .
    type File_Descriptor is private;
    . . .
private
    type File_Descriptor is new Integer;
end OS;

package OS.Exceptions is
    File_Descriptor_Error,
    File_Name_Error,
    Permission_Error : exception;
end OS.Exceptions;

with OS.Exceptions;
package OS.File_Manager is
    type File_Mode is ( Read_Only, Write_Only, Read_Write);
    function Open( File_Name : String; Mode : File_Mode) return
    OS.File_Descriptor;
    procedure Close ( File : in OS.File_Descriptor );
    . . .
end OS.File_Manager;
```

```
procedure OS.Interpret ( Command : String );


private package OS.Internals is
    . . .
end OS.Internals;


private package OS.Internals_Debug is
    . . .
end OS.Internals_Debug;
```

OS.Internals and OS.Internals_Debug are both private child packages of the OS

Two visibility rules we have to mention extra than the other occur with the general child units.

► The private child is only visible within the subtree of the hierarchy whose root is its parent. And moreover within that tree it is not visible to the specifications of any non-private siblings.

OS.Internals private child is visible to the bodies of OS itself, of OS.File_Manager and of OS.Interpret and it is also visible to both body and specification of OS.Internal_Debug. It is not visible outside OS and a client package certainly cannot access OS.Internals at all.

► The visible part of the private child can access the private part of its parent. So it cannot export information about the private type to a client because it is not itself visible. Nor can it export information indirectly via its non-private siblings because, as we have seen, it is not visible to their specifications but only to their bodies.

**Generic Children**

If the parent unit is a non generic unit then the children unit maybe generic or not.

If the parent unit is generic then all the children must always be generic.

The main problems that must be solved in the design of the interaction between genericity and children is the impact of new children on existing instantiations.

A generic child of a generic parent can be instantiated inside the parent and its hierarchy ( as normal ) or externally but then only as a child of an instantiation of its parent.

```
generic
    type T is private;
package Parent is
    . . .
end Parent;

generic
package Parent.Child is
    . . .
end Parent.Child;
```

and the typical instantiations might be

```
with Parent;
package Instance is new Parent ( T => Integer);

with Parent.Child;
package Instance.Child is new Parent.Child;
```

Instantiations inside the parent hierarchy poses no problem since the instantiation has visibility of the parent's formal parameters in the usual way.

Instantiations outside requires that the actual parameter corresponding to the formal parameter is correspondingly visible to the instantiation of the child. This is assured by insisting that the child is only instantiated as a child of an instance of the parent as in the example.

**Generics**

➢ We can use a formal parameter notation for unconstrained types for example

**type** T(<>) **is private;**

We allow to declare an ( uninitialized ) object of type T in the generic body.

T is used only when we do not require a constrained type.

➢ The notation

**type** T **is private;**

is used when T is Integer or a constrained type or a record type with default discriminants.

➢ The formal parameter can be

**type** T **is tagged private;**

Requires the actual type is tagged (not abstract).

➢ We can write the notations

**type** T **is new S;** or

**type** T **is new** S **with private;**

The actual type must be S or derived directly or indirectly from S.

In the second case, with private indicates that the actual type must also be tagged.

```
generic
    type S is tagged private;
package P is
    type T is new S with private;
    -- operations on T
private
    type T is new S with
        record
        -- additional components
        end record;
end P;
```

➢ We instantiate the package P to add the operations of T to any existing tagged type.

The resulting type will of course still be in the class of the type passed as actual parameter.

```
        type Shape is new Object with private;
```

We can extend the Shape type privately with generic operations.

```
private
        package Q is new P(Object);
        type Shape is new Q.T with null record;
```

## LIST OF REFERENCES

1. Ada 9X Mapping / Revision Team " Ada 9X Reference Manual, The Language, The standard Libraries " , June 1994.
2. Ada 9X Mapping / Revision Team " Ada 9X Rationale The Language, The Stan dard Libraries " , June 1994.
3. Ada 9X Mapping / Revision Team "Changes from Ada 83 to Ada 9X", July 1994.
4. Anderson M. Christine article " Ada 9X Project Report to the Public" in Cross Talk THE DEFENSE SOFTWARE ENGINEERING REPORT , March 1994.
5. Anderson M. Christine article " Ada 83 / Ada 9X Compatibility" in Cross Talk THE DEFENSE SOFTWARE ENGINEERING REPORT , October 1992.
6. Barnes J.G.P. Programming in Ada Plus an Overview of Ada 9X , 1994.
7. Budd Timothy " An Introduction to Object Oriented Programming , 1991.
8. DaCosta Robert Defense Science Magazine Article "The History of Ada" , March 1984.
9. Feldman B. Michael , Koffman B. Elliot Ada Problem Solving and Program Design, 1993.
10. Gonzalez Dean "Ada Programmer's handbook" , June 1991.
11. Ploedereder Erhard , How to Program in Ada 94, Using Ada 83, 1992.
12. Sammet Jean E. " Why Ada Is Not Just Another Programming Language " Communications of the ACM, August 1986.
13. Skansholm Jan , Ada From the Beginning Second Edition, 1994.

## INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center ........................................2
   Cameron Station
   Alexandria, VA 22304-6145

2. Dudley Knox Library ........................................2
   Code 52
   Naval Postgraduate School
   Monterey, CA 93943-5101

3. Chairman, Code CS........................................1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

4. LT Col. David A. Gaitros, Code CS/Gi ........................................1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

5. Lucia Luqi, Code CS/Lq........................................1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. LT John T. Drimousis........................................2
   Harilaou Trikoupi 9
   Salamis, Athens